CSS first steps

CSS (Cascading Style Sheets) is used to style and lay out web pages — for example, to alter the font, color, size, and spacing of your content, split it into multiple columns, or add animations and other decorative features. This module provides a gentle beginning to your path towards CSS mastery with the basics of how it works, what the syntax looks like, and how you can start using it to add styling to HTML.

Prerequisites

mdn web docs

1. Basic familiarity with using computers and using the Web passively (i.e. looking at it, consuming the content.)
2. A basic work environment set up, as detailed in Installing basic software, and an understanding of how to create and manage files, as detailed in Dealing with files.
3. Basic familiarity with HTML, as discussed in the Introduction to HTML module.

> **Note:** If you are working on a computer/tablet/other device where you don't have the ability to create your own files, you could try out (most of) the code examples in an online coding program such as JSBin or Glitch .

Guides

This module contains the following articles, which will take you through all the basic theory of CSS, and provide opportunities for you to test out some skills.

What is CSS?

CSS (Cascading Style Sheets) allows you to create great-looking web pages, but how does it work under the hood? This article explains what CSS is with a simple syntax example and also covers some key terms about the language.

Getting started with CSS

In this article, we will take a simple HTML document and apply CSS to it, learning some practical things about the language along the way.

How CSS is structured

Now that you have an idea about what CSS is and the basics of using it, it is time to look a little deeper into the structure of the language itself. We have already met many of the concepts discussed here; you can return to this one to recap if you find any later concepts confusing.

How CSS works

We have learned the basics of CSS, what it is for, and how to write simple stylesheets. In this article, we will take a look at how a browser takes CSS and HTML and turns that into a webpage.

Assessments

The following assessment will test your understanding of the CSS basics covered in the guides above.

[Styling a biography page]

With the things you have learned in the last few articles, you should find that you can format simple text documents using CSS to add your own style to them. This assessment gives you a chance to do that.

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Mar 12, 2024 by MDN contributors.

CSS selectors

In CSS, selectors are used to target the HTML elements on our web pages that we want to style. There are a wide variety of CSS selectors available, allowing for fine-grained precision when selecting elements to style. In this article and its sub-articles we'll run through the different types in great detail, seeing how they work.

Prerequisites:	Basic software installed, basic knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps.)
Objective:	To learn how CSS selectors work in detail.

What is a selector?

A CSS selector is the first part of a CSS Rule. It is a pattern of elements and other terms that tell the browser which HTML elements should be selected to have the CSS property values inside the rule applied to them. The element or elements which are selected by the selector are referred to as the *subject of the selector*.

```
h1 {
  color: blue;
  background-color: yellow;
}

p {
  color: red;
}
```

In other articles you may have met some different selectors, and learned that there are selectors that target the document in different ways — for example by selecting an element such as `h1`, or a class such as `.special`.

In CSS, selectors are defined in the CSS Selectors specification; like any other part of CSS they need to have support in browsers for them to work. The majority of selectors that you will come across are defined in the Level 3 Selectors specification and Level 4 Selectors specification, which are both mature specifications, therefore you will find excellent browser support for these selectors.

Selector lists

If you have more than one thing which uses the same CSS then the individual selectors can be combined into a *selector list* so that the rule is applied to all of the individual selectors. For example, if I have the same CSS for an `h1` and also a class of `.special`, I could write this as two separate rules.

CSS

```
h1 {
  color: blue;
}

.special {
  color: blue;
}
```

I could also combine these into a selector list, by adding a comma between them.

```css
h1, .special {
  color: blue;
}
```

White space is valid before or after the comma. You may also find the selectors more readable if each is on a new line.

```css
h1,
.special {
  color: blue;
}
```

In the live example below try combining the two selectors which have identical declarations. The visual display should be the same after combining them.

Type selectors

Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon azuki bean garlic.

Gumbo beet greens corn soko **endive** gumbo gourd. Parsley shallot courgette tatsoi pea sprouts fava bean collard greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut soko zucchini.

Turnip greens yarrow ricebean rutabaga *endive cauliflower* sea lettuce kohlrabi amaranth water spinach avocado daikon napa cabbage asparagus winter purslane kale. Celery potato scallion desert raisin horseradish spinach

Interactive editor

```
span {
  background-color: yellow;
}

strong {
  color: rebeccapurple;
}

em {
  color: rebeccapurple;
}
```

```
<h1>Type selectors</h1>
<p>Veggies es bonus vobis, proinde vos postulo essum magis <span>kohlrabi welsh
onion</span> daikon amaranth tatsoi tomatillo
    melon azuki bean garlic.</p>

<p>Gumbo beet greens corn soko <strong>endive</strong> gumbo gourd. Parsley
shallot courgette tatsoi pea sprouts fava bean collard
    greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut
soko zucchini.</p>

<p>Turnip greens yarrow ricebean rutabaga <em>endive cauliflower</em> sea lettuce
kohlrabi amaranth water spinach avocado
    daikon napa cabbage asparagus winter purslane kale. Celery potato scallion
desert raisin horseradish spinach
</p>
```

Reset

When you group selectors in this way, if any selector is syntactically invalid, the whole rule will be ignored.

In the following example, the invalid class selector rule will be ignored, whereas the `h1` would still be styled.

CSS

```
h1 {
  color: blue;
}

..special {
```

```
  color: blue;
}
```

When combined however, neither the `h1` nor the class will be styled as the entire rule is deemed invalid.

```css
h1, ..special {
  color: blue;
}
```

Types of selectors

There are a few different groupings of selectors, and knowing which type of selector you might need will help you to find the right tool for the job. In this article's subarticles we will look at the different groups of selectors in more detail.

Type, class, and ID selectors

Type selectors target an HTML element such as an `<h1>`:

```css
h1 {
}
```

Class selectors target an element that has a specific value for its `class` attribute:

```css
.box {
}
```

ID selectors target an element that has a specific value for its `id` attribute:

```css
#unique {
}
```

Attribute selectors

This group of selectors gives you different ways to select elements based on the presence of a certain attribute on an element:

```css
a[title] {
}
```

Or even make a selection based on the presence of an attribute with a particular value:

```css
a[href="https://example.com"]
{
}
```

Pseudo-classes and pseudo-elements

This group of selectors includes pseudo-classes, which style certain states of an element. The `:hover` pseudo-class for example selects an element only when it is being hovered over by the mouse pointer:

CSS

```
a:hover {
}
```

It also includes pseudo-elements, which select a certain part of an element rather than the element itself. For example, `::first-line` always selects the first line of text inside an element (a `<p>` in the below case), acting as if a `<span>` was wrapped around the first formatted line and then selected.

CSS

```
p::first-line {
}
```

Combinators

The final group of selectors combine other selectors in order to target elements within our documents. The following, for example, selects paragraphs that are direct children of `<article>` elements using the child combinator (`>`):

CSS

```
article > p {
}
```

Summary

In this article we've introduced CSS selectors, which enable you to target particular HTML elements. Next, we'll take a closer look at type, class, and ID selectors.

For a complete list of selectors, see our CSS selectors reference.

Type, class, and ID selectors

In this lesson, we examine some of the simplest selectors, which you will probably use most frequently in your work.

Prerequisites:	Basic software installed, basic knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps.)
Objective:	To learn about the different CSS selectors we can use to apply CSS to a document.

Type selectors

A **type selector** is sometimes referred to as a *tag name selector* or *element selector* because it selects an HTML tag/element in your document. Type selectors are not case-sensitive. In the example below, we have used the `span`, `em` and `strong` selectors.

Try adding a CSS rule to select the `<h1>` element and change its color to blue.

Type selectors

Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon azuki bean garlic.

Gumbo beet greens corn soko **endive** gumbo gourd. Parsley shallot courgette tatsoi pea sprouts fava bean collard greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut soko zucchini.

Turnip greens yarrow ricebean rutabaga *endive cauliflower* sea lettuce kohlrabi amaranth water spinach avocado daikon napa cabbage asparagus winter purslane kale. Celery potato scallion desert raisin horseradish spinach

Interactive editor

```
span {
  background-color: yellow;
}

strong {
  color: rebeccapurple;
}

em {
  color: rebeccapurple;
}
```

```
<h1>Type selectors</h1>
<p>Veggies es bonus vobis, proinde vos postulo essum magis <span>kohlrabi welsh
onion</span> daikon amaranth tatsoi tomatillo
    melon azuki bean garlic.</p>

<p>Gumbo beet greens corn soko <strong>endive</strong> gumbo gourd. Parsley
shallot courgette tatsoi pea sprouts fava bean collard
    greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut
soko zucchini.</p>

<p>Turnip greens yarrow ricebean rutabaga <em>endive cauliflower</em> sea lettuce
kohlrabi amaranth water spinach avocado
    daikon napa cabbage asparagus winter purslane kale. Celery potato scallion
desert raisin horseradish spinach
</p>
```

Reset

The universal selector

The universal selector is indicated by an asterisk (`*`). It selects everything in the document (or inside the parent element if it is being chained together with another element and a descendant combinator). In the following example, we use the universal selector to remove the margins on all elements. Instead of the default styling added by the browser — which spaces out headings and paragraphs with margins — everything is close together.

Universal selector

Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon azuki bean garlic.
Gumbo beet greens corn soko **endive** gumbo gourd. Parsley shallot courgette tatsoi pea sprouts fava bean collard greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut soko zucchini.

Interactive editor

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

```
<h1>Universal selector</h1>
<p>Veggies es bonus vobis, proinde vos postulo essum magis <span>kohlrabi welsh
onion</span> daikon amaranth tatsoi tomatillo
    melon azuki bean garlic.</p>

<p>Gumbo beet greens corn soko <strong>endive</strong> gumbo gourd. Parsley
shallot courgette tatsoi pea sprouts fava bean collard
    greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut
soko zucchini.</p>
```

Reset

This kind of behavior can sometimes be seen in "reset stylesheets", which strip out all of the browser styling. Since the universal selector makes global changes, we use it for very specific situations, such as the one described below.

Using the universal selector to make your selectors easier to read

One use of the universal selector is to make selectors easier to read and more obvious in terms of what they are doing. For example, if we wanted to select any descendant elements of an `<article>` element that are the first child of their parent, including direct children, and make them bold, we could use the `:first-child` pseudo-class. We will learn more about this in the lesson on pseudo-classes and pseudo-elements, as a descendant selector along with the `<article>` element selector:

CSS

```
article :first-child {
  font-weight: bold;
}
```

However, this selector could be confused with `article:first-child`, which will select any `<article>` element that is the first child of another element.

To avoid this confusion, we can add the universal selector to the `:first-child` pseudo-class, so it is more obvious what the selector is doing. It is selecting *any* element which is the first-child of an `<article>` element, or the first-child of any descendant element of `<article>`:

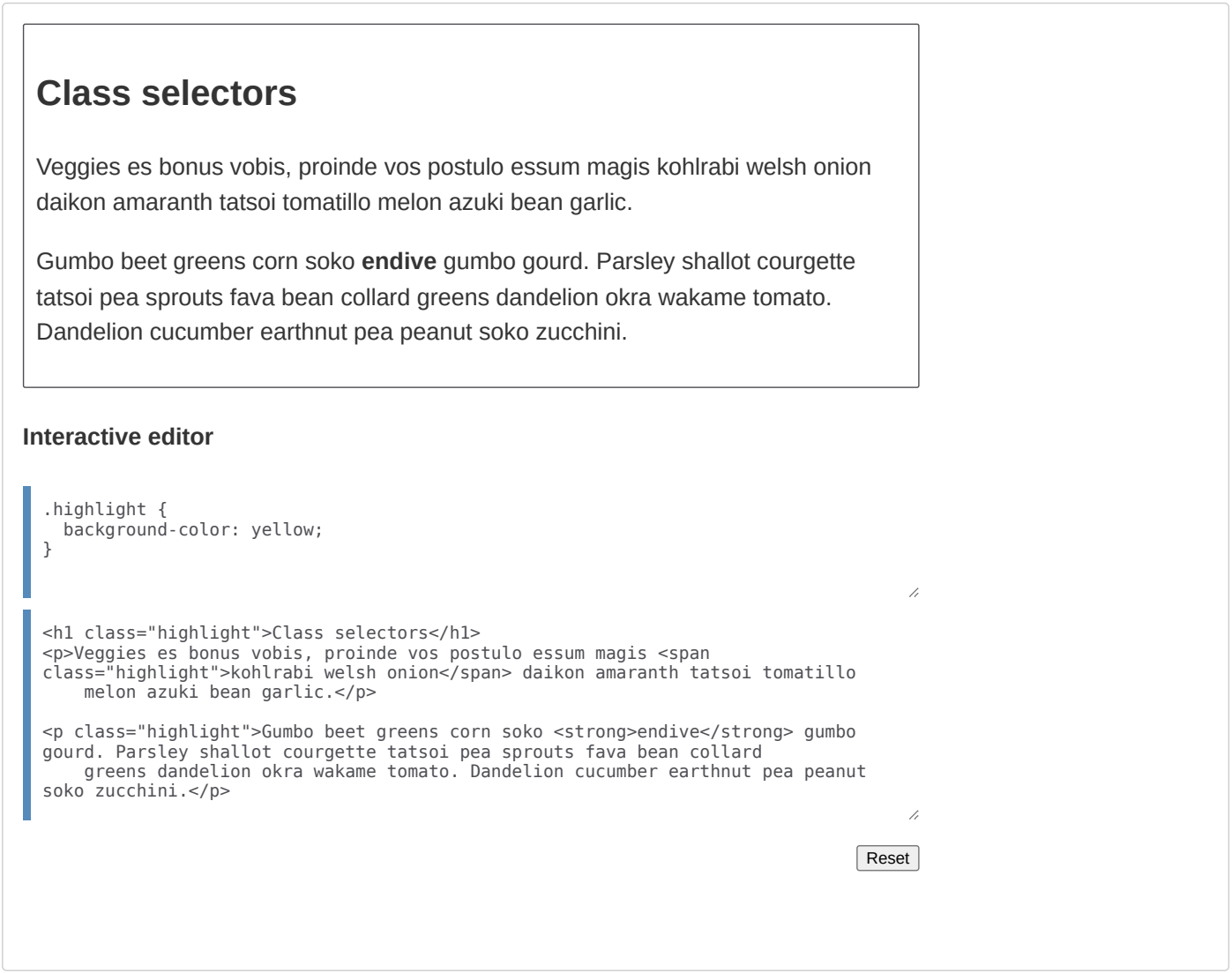
CSS

```
article *:first-child {
  font-weight: bold;
}
```

Although both do the same thing, the readability is significantly improved.

Class selectors

The case-sensitive class selector starts with a dot (.) character. It will select everything in the document with that class applied to it. In the live example below we have created a class called `highlight`, and have applied it to several places in my document. All of the elements that have the class applied are highlighted.



Targeting classes on particular elements

You can create a selector that will target specific elements with the class applied. In this next example, we will highlight a `<span>` with a class of `highlight` differently to an `<h1>` heading with a class of `highlight`. We do this by using the type selector for the element we want to target, with the class appended using a dot, with no white space in between.

Class selectors

Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon azuki bean garlic.

Gumbo beet greens corn soko **endive** gumbo gourd. Parsley shallot courgette tatsoi pea sprouts fava bean collard greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut soko zucchini.

Interactive editor

```
span.highlight {
  background-color: yellow;
}

h1.highlight {
  background-color: pink;
}
```

```
<h1 class="highlight">Class selectors</h1>
<p>Veggies es bonus vobis, proinde vos postulo essum magis <span
class="highlight">kohlrabi welsh onion</span> daikon amaranth tatsoi tomatillo
    melon azuki bean garlic.</p>

<p class="highlight">Gumbo beet greens corn soko <strong>endive</strong> gumbo
gourd. Parsley shallot courgette tatsoi pea sprouts fava bean collard
    greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut
soko zucchini.</p>
```

Reset

This approach reduces the scope of a rule. The rule will only apply to that particular element and class combination. You would need to add another selector if you decided the rule should apply to other elements too.

Target an element if it has more than one class applied

You can apply multiple classes to an element and target them individually, or only select the element when all of the classes in the selector are present. This can be helpful when building up components that can be combined in different ways on your site.

In the example below, we have a `<div>` that contains a note. The grey border is applied when the box has a class of `notebox`. If it also has a class of `warning` or `danger`, we change the `border-color`.

We can tell the browser that we only want to match the element if it has two classes applied by chaining them together with no white space between them. You'll see that the last `<div>` doesn't get any styling applied, as it only has the `danger` class; it needs `notebox` as well to get anything applied.

mdn web docs_

Interactive editor

```
.notebox {
  border: 4px solid #666;
  padding: .5em;
}

.notebox.warning {
  border-color: orange;
  font-weight: bold;
}

.notebox.danger {
  border-color: red;
  font-weight: bold;
}
```

```
<div class="notebox">
    This is an informational note.
</div>

<div class="notebox warning">
    This note shows a warning.
</div>

<div class="notebox danger">
    This note shows danger!
</div>

<div class="danger">
    This won't get styled — it also needs to have the notebox class
</div>
```

Reset

ID selectors

The case-sensitive ID selector begins with a `#` rather than a dot character, but is used in the same way as a class selector. However, an ID can be used only once per page, and elements can only have a single `id` value applied to them. It can select an element that has the `id` set on it, and you can precede the ID with a type selector to only target the element if both the element and ID match. You can see both of these uses in the following example:

ID selector

Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon azuki bean garlic.

Gumbo beet greens corn soko **endive** gumbo gourd. Parsley shallot courgette tatsoi pea sprouts fava bean collard greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut soko zucchini.

Interactive editor

```
#one {
  background-color: yellow;
}

h1#heading {
  color: rebeccapurple;
}
```

```
<h1 id="heading">ID selector</h1>
<p>Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion
daikon amaranth tatsoi tomatillo
    melon azuki bean garlic.</p>

<p id="one">Gumbo beet greens corn soko <strong>endive</strong> gumbo gourd.
Parsley shallot courgette tatsoi pea sprouts fava bean collard
    greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut
soko zucchini.</p>
```

Reset

Warning: Using the same ID multiple times in a document may appear to work for styling purposes, but don't do this. It results in invalid code, and will cause strange behavior in many places.

Note: The ID selector has high `specificity`. This means styles applied based on matching an ID selector will overrule styles applied based on other selector, including class and type selectors. Because an ID can only occur once on a page and because of the high specificity of ID selectors, it is preferable to add a class to an element instead of an ID. If using the ID is the only way to target the element — perhaps because you do not have access to the markup and cannot edit it — consider using the ID within an attribute selector, such as `p[id="header"]`. Learn specificity.

Summary

That wraps up Type, class, and ID selectors. We'll continue exploring selectors by looking at attribute selectors.

Attribute selectors

As you know from your study of HTML, elements can have attributes that give further detail about the element being marked up. In CSS you can use attribute selectors to target elements with certain attributes. This lesson will show you how to use these very useful selectors.

Prerequisites:	Basic software installed, basic knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps.)
Objective:	To learn what attribute selectors are and how to use them.

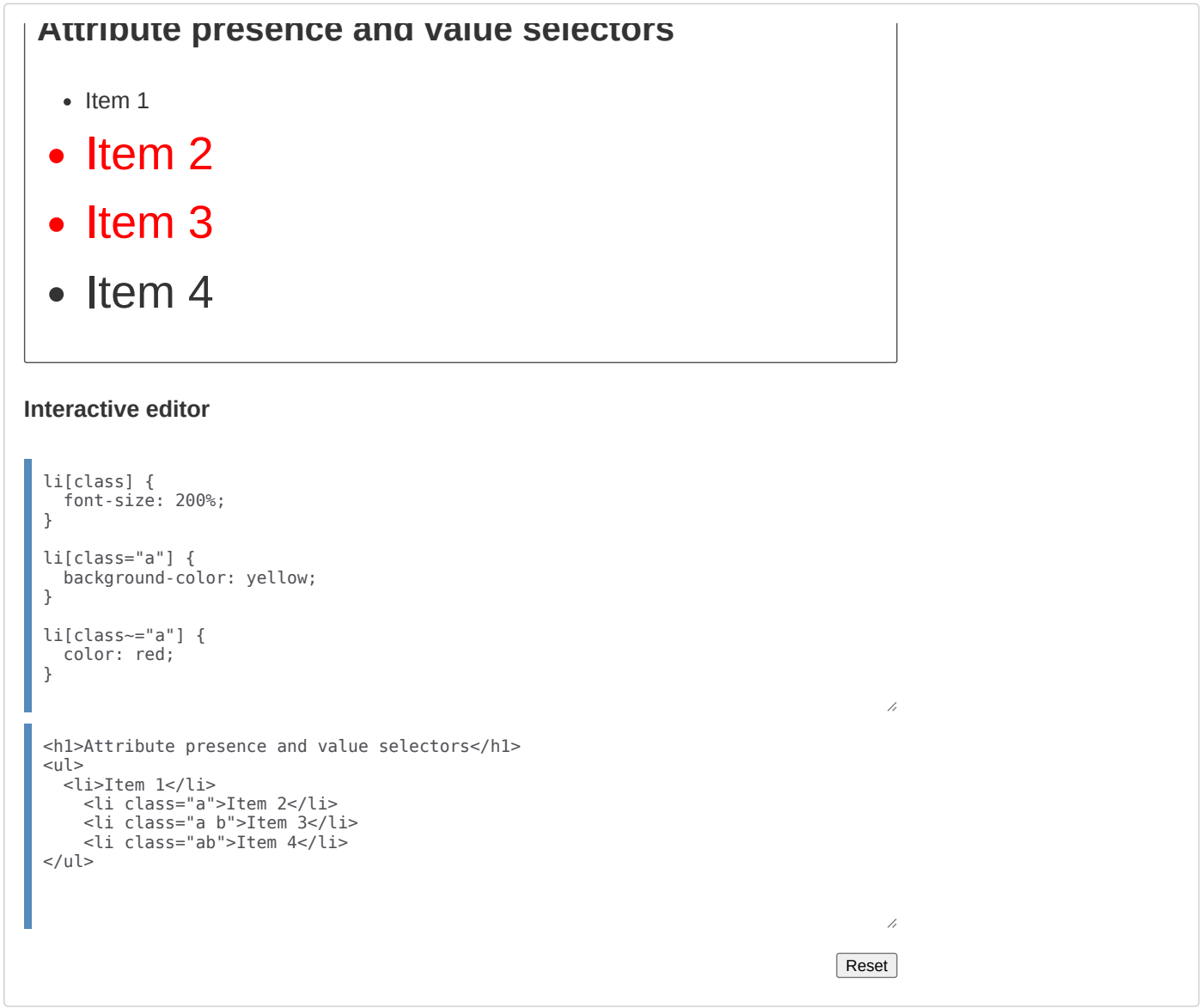
Presence and value selectors

These selectors enable the selection of an element based on the presence of an attribute alone (for example `href`), or on various different matches against the value of the attribute.

Selector	Example	Description
`[attr]`	`a[title]`	Matches elements with an *attr* attribute (whose name is the value in square brackets).
`[attr=value]`	`a[href="https://example.com"]`	Matches elements with an *attr* attribute whose value is exactly *value* — the string inside the quotes.
`[attr~=value]`	`p[class~="special"]`	Matches elements with an *attr* attribute whose value is exactly *value*, or contains *value* in its (space separated) list of values.
`[attr|=value]`	`div[lang|="zh"]`	Matches elements with an *attr* attribute whose value is exactly *value* or begins with *value* immediately followed by a hyphen.

In the example below you can see these selectors being used.

- By using `li[class]` we can match any list item with a class attribute. This matches all of the list items except the first one.
- `li[class="a"]` matches a selector with a class of `a`, but not a selector with a class of `a` with another space-separated class as part of the value. It selects the second list item.
- `li[class~="a"]` will match a class of `a` but also a value that contains the class of `a` as part of a whitespace-separated list. It selects the second and third list items.



Substring matching selectors

These selectors allow for more advanced matching of substrings inside the value of your attribute. For example, if you had classes of `box-warning` and `box-error` and wanted to match everything that started with the string "box-", you could use `[class^="box-"]` to select them both (or `[class|="box"]` as described in section above).

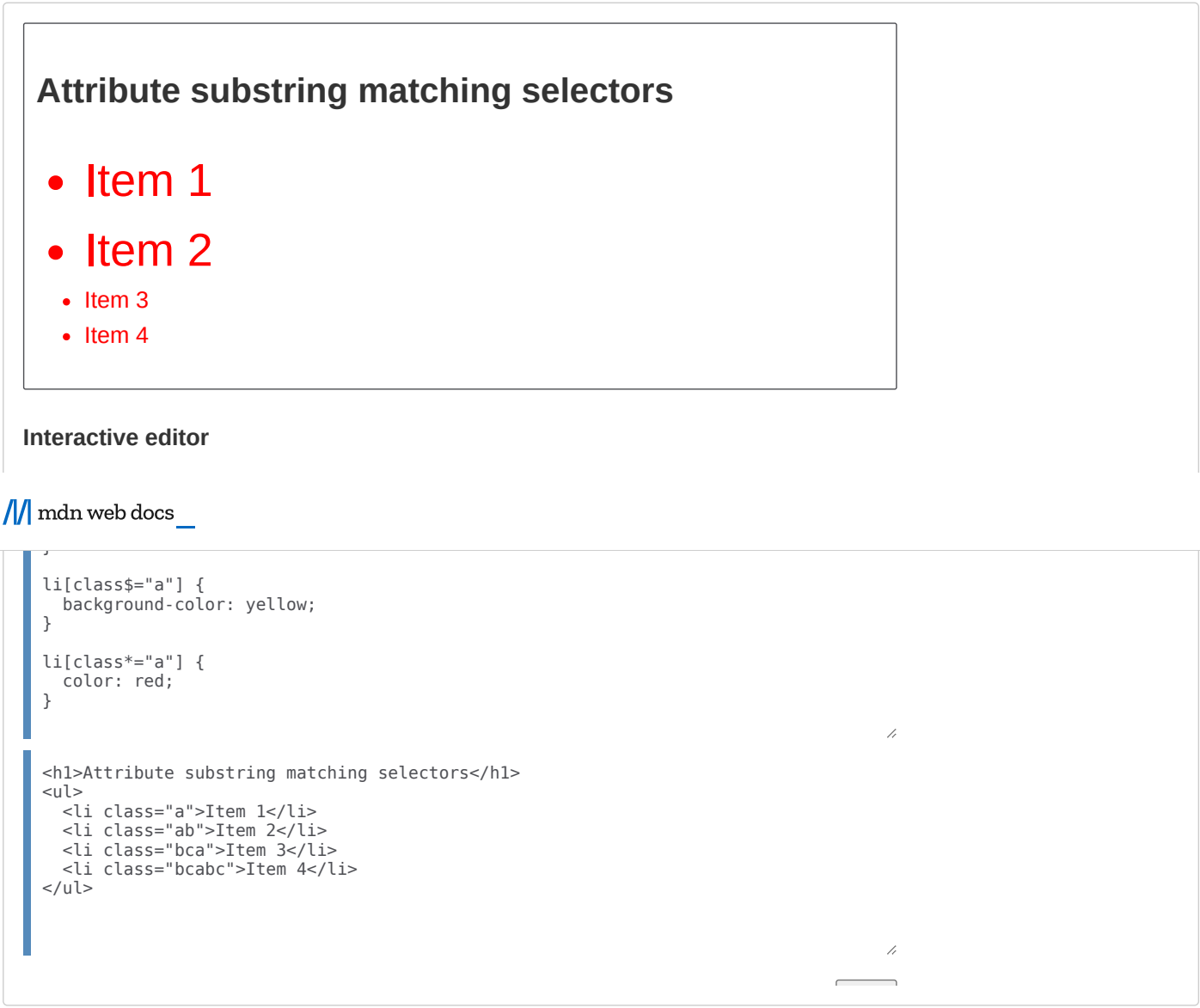
Selector	Example	Description
`[attr^=value]`	`li[class^="box-"]`	Matches elements with an *attr* attribute, whose value begins with *value*.
`[attr$=value]`	`li[class$="-box"]`	Matches elements with an *attr* attribute whose value ends with *value*.
`[attr*=value]`	`li[class*="box"]`	Matches elements with an *attr* attribute whose value contains *value* anywhere within the string.

(Aside: It may help to note that `^` and `$` have long been used as *anchors* in so-called *regular expressions* to mean *begins with* and *ends with* respectively.)

The next example shows usage of these selectors:

- `li[class^="a"]` matches any attribute value which starts with `a`, so matches the first two list items.

- `li[class$="a"]` matches any attribute value that ends with `a`, so matches the first and third list item.
- `li[class*="a"]` matches any attribute value where `a` appears anywhere in the string, so it matches all of our list items.



Case-sensitivity

If you want to match attribute values case-insensitively you can use the value `i` before the closing bracket. This flag tells the browser to match ASCII characters case-insensitively. Without the flag the values will be matched according to the case-sensitivity of the document language — in HTML's case it will be case sensitive.

In the example below, the first selector will match a value that begins with `a` — it only matches the first list item because the other two list items start with an uppercase A. The second selector uses the case-insensitive flag and so matches all of the list items.

Case-insensitivity

- Item 1
- Item 2
- Item 3

Interactive editor

```
li[class^="a"] {
  background-color: yellow;
}

li[class^="a" i] {
  color: red;
}
```

```
<h1>Case-insensitivity</h1>
<ul>
  <li class="a">Item 1</li>
  <li class="A">Item 2</li>
  <li class="Ab">Item 3</li>
</ul>
```

Reset

> **Note:** There is also a newer value `s`, which will force case-sensitive matching in contexts where matching is normally case-insensitive, however this is less well supported in browsers and isn't very useful in an HTML context.

Summary

Now that we are done with attribute selectors, you can continue on to the next article and read about pseudo-class and pseudo-element selectors.

The box model

Everything in CSS has a box around it, and understanding these boxes is key to being able to create more complex layouts with CSS, or to align items with other items. In this lesson, we will take a look at the CSS *Box Model*. You'll get an understanding of how it works and the terminology that relates to it.

Prerequisites:	Basic software installed, basic knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps.)
Objective:	To learn about the CSS Box Model, what makes up the box model and how to switch to the alternate model.

Block and inline boxes

In CSS we have several types of boxes that generally fit into the categories of **block boxes** and **inline boxes**. The type refers to how the box behaves in terms of page flow and in relation to other boxes on the page. Boxes have an **inner display type** and an **outer display type**.

In general, you can set various values for the display type using the `display` property, which can have various values.

Outer display type

If a box has an outer display type of `block`, then:

- The box will break onto a new line.
- The `width` and `height` properties are respected.
- Padding, margin and border will cause other elements to be pushed away from the box.
- If `width` is not specified, the box will extend in the inline direction to fill the space available in its container. In most cases, the box will become as wide as its container, filling up 100% of the space available.

Some HTML elements, such as `<h1>` and `<p>`, use `block` as their outer display type by default.

If a box has an outer display type of `inline`, then:

- The box will not break onto a new line.
- The `width` and `height` properties will not apply.
- Top and bottom padding, margins, and borders will apply but will not cause other inline boxes to move away from the box.
- Left and right padding, margins, and borders will apply and will cause other inline boxes to move away from the box.

Some HTML elements, such as `<a>`, `<span>`, `<em>` and `<strong>` use `inline` as their outer display type by default.

Inner display type

Boxes also have an *inner* display type, which dictates how elements inside that box are laid out.

Block and inline layout is the default way things behave on the web. By default and without any other instruction, the elements inside a box are also laid out in **normal flow** and behave as block or inline boxes.

You can change the inner display type for example by setting `display: flex;`. The element will still use the outer display type `block` but this changes the inner display type to `flex`. Any direct children of this box will become flex items and behave according to the Flexbox specification.

When you move on to learn about CSS Layout in more detail, you will encounter `flex`, and various other inner values that your boxes can have, for example `grid`.

> **Note:** To read more about the values of display, and how boxes work in block and inline layout, take a look at the MDN guide Block and Inline Layout.

Examples of different display types

The example below has three different HTML elements, all of which have an outer display type of `block`.

- A paragraph with a border added in CSS. The browser renders this as a block box. The paragraph starts on a new line and extends the entire available width.
- A list, which is laid out using `display: flex`. This establishes flex layout for the children of the container, which are flex items. The list itself is a block box and — like the paragraph — expands to the full container width and breaks onto a new line.
- A block-level paragraph, inside which are two `<span>` elements. These elements would normally be `inline`, however, one of the elements has a class of "block" which gets set to `display: block`.

I am a paragraph. A short one.

Item One Item Two Item Three

I am another paragraph. Some of the

words

have been wrapped in a span element.

Interactive editor

```
p,
ul {
  border: 2px solid rebeccapurple;
  padding: .5em;
}

.block,
li {
  border: 2px solid blue;
  padding: .5em;
}

ul {
  display: flex;
  list-style: none;
}

.block {
  display: block;
}
```

```
<p>I am a paragraph. A short one.</p>
<ul>
  <li>Item One</li>
  <li>Item Two</li>
  <li>Item Three</li>
</ul>
<p>I am another paragraph. Some of the <span class="block">words</span> have been
wrapped in a <span>span element</span>.</p>
```

Reset

In the next example, we can see how `inline` elements behave.

- The `<span>` elements in the first paragraph are inline by default and so do not force line breaks.
- The `<ul>` element that is set to `display: inline-flex` creates an inline box containing some flex items.
- The two paragraphs are both set to `display: inline`. The inline flex container and paragraphs all run together on one line rather than breaking onto new lines (as they would do if they were displaying as block-level elements).

To toggle between the display modes, you can change `display: inline` to `display: block` or `display: inline-flex` to `display: flex`.

I am a paragraph. Some of the words have been wrapped in a span element.

Item One Item Two Item Three I am a paragraph. A short one. I am another paragraph. Also a short one.

Interactive editor

```
p,
ul {
  border: 2px solid rebeccapurple;
}

span,
li {
  border: 2px solid blue;
}

ul {
  display: inline-flex;
  list-style: none;
  padding: 0;
}

.inline {
  display: inline;
}
```

```
<p>
  I am a paragraph. Some of the
  <span>words</span> have been wrapped in a
  <span>span element</span>.
</p>
<ul>
  <li>Item One</li>
  <li>Item Two</li>
  <li>Item Three</li>
</ul>
<p class="inline">I am a paragraph. A short one.</p>
<p class="inline">I am another paragraph. Also a short one.</p>
```

Reset

The key thing to remember for now is: Changing the value of the `display` property can change whether the outer display type of a box is block or inline. This changes the way it displays alongside other elements in the layout.

What is the CSS box model?

The CSS box model as a whole applies to block boxes and defines how the different parts of a box — margin, border, padding, and content — work together to create a box that you can see on a page. Inline boxes use just *some* of the behavior defined in the box model.

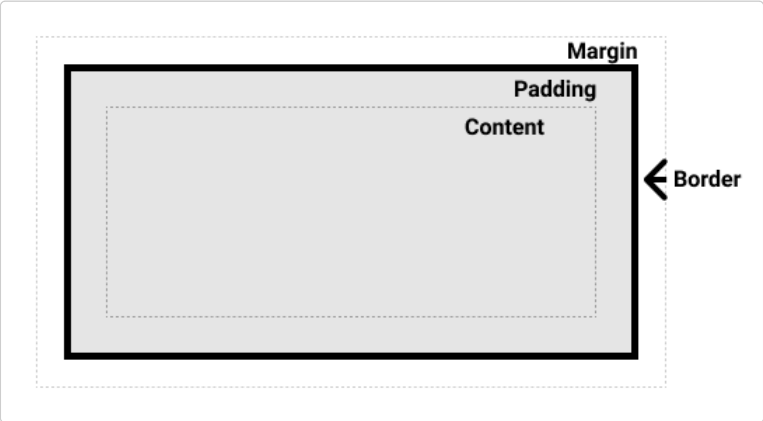
To add complexity, there is a standard and an alternate box model. By default, browsers use the standard box model.

Parts of a box

Making up a block box in CSS we have the:

- **Content box**: The area where your content is displayed; size it using properties like `inline-size` and `block-size` or `width` and `height`.
- **Padding box**: The padding sits around the content as white space; size it using `padding` and related properties.
- **Border box**: The border box wraps the content and any padding; size it using `border` and related properties.
- **Margin box**: The margin is the outermost layer, wrapping the content, padding, and border as whitespace between this box and other elements; size it using `margin` and related properties.

The below diagram shows these layers:



The standard CSS box model

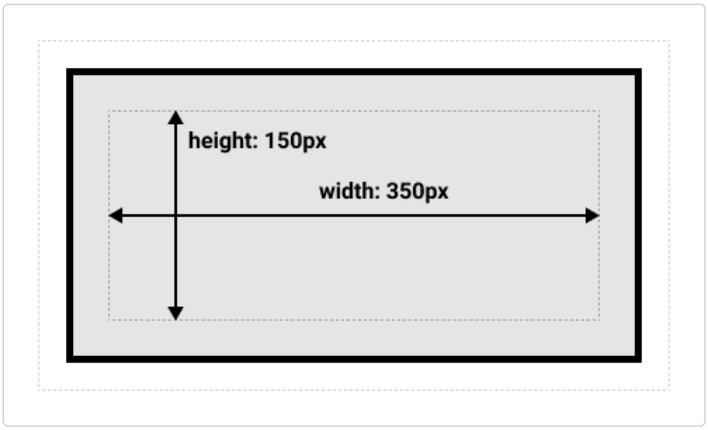
In the standard box model, if you set `inline-size` and `block-size` (or `width` and `height`) property values on a box, these values define the `inline-size` and `block-size` (`width` and `height` in horizontal languages) of the *content box*. Any padding and borders are then added to those dimensions to get the total size taken up by the box (see the image below).

If we assume that a box has the following CSS:

CSS

```css
.box {
  width: 350px;
  height: 150px;
  margin: 10px;
  padding: 25px;
  border: 5px solid black;
}
```

The *actual* space taken up by the box will be 410px wide (350 + 25 + 25 + 5 + 5) and 210px high (150 + 25 + 25 + 5 + 5).



> **Note:** The margin is not counted towards the actual size of the box — sure, it affects the total space that the box will take up on the page, but only the space outside the box. The box's area stops at the border — it does not extend into the margin.

The alternative CSS box model

In the alternative box model, any width is the width of the visible box on the page. The content area width is that width minus the width for the padding and border (see image below). No need to add up the border and padding to get the real size of the box.

To turn on the alternative model for an element, set `box-sizing: border-box` on it:

CSS

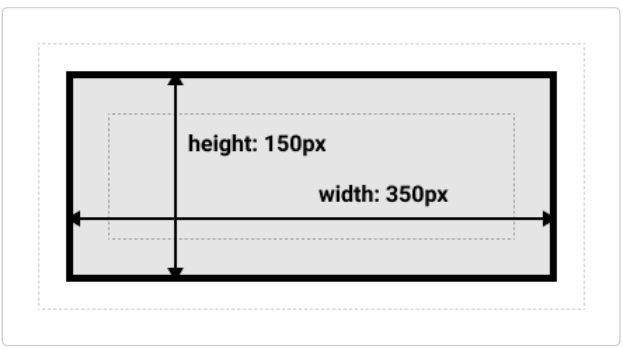
```css
.box {
  box-sizing: border-box;
}
```

If we assume the box has the same CSS as above:

CSS

```css
.box {
  width: 350px;
  inline-size: 350px;
  height: 150px;
  block-size: 150px;
  margin: 10px;
  padding: 25px;
  border: 5px solid black;
}
```

Now, the *actual* space taken up by the box will be 350px in the inline direction and 150px in the block direction.



To use the alternative box model for all of your elements (which is a common choice among developers), set the `box-sizing` property on the `<html>` element and set all other elements to inherit that value:

CSS

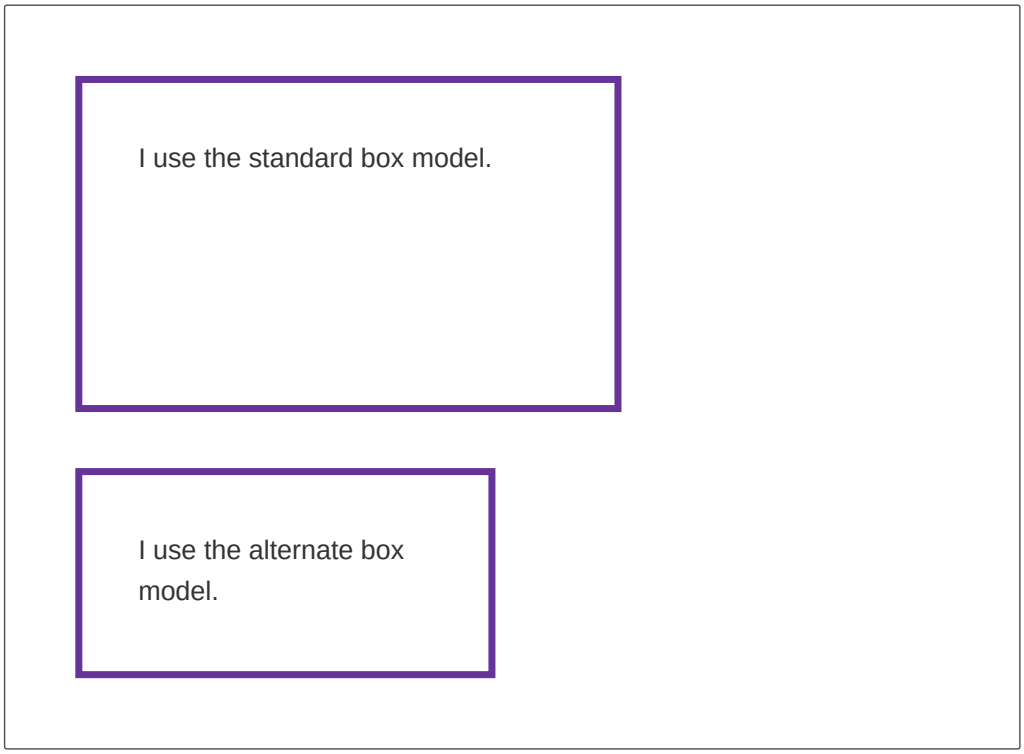
```
html {
  box-sizing: border-box;
}

*,
*::before,
*::after {
  box-sizing: inherit;
}
```

To understand the underlying idea, you can read the CSS Tricks article on box-sizing .

Playing with box models

In the example below, you can see two boxes. Both have a class of `.box`, which gives them the same `width`, `height`, `margin`, `border`, and `padding`. The only difference is that the second box has been set to use the alternative box model.

Can you change the size of the second box (by adding CSS to the `.alternate` class) to make it match the first box in width and height?



Interactive editor

```
.box {
  border: 5px solid rebeccapurple;
  background-color: lightgray;
  padding: 40px;
  margin: 40px;
  width: 300px;
  height: 150px;
}

.alternate {
  box-sizing: border-box;
}
```

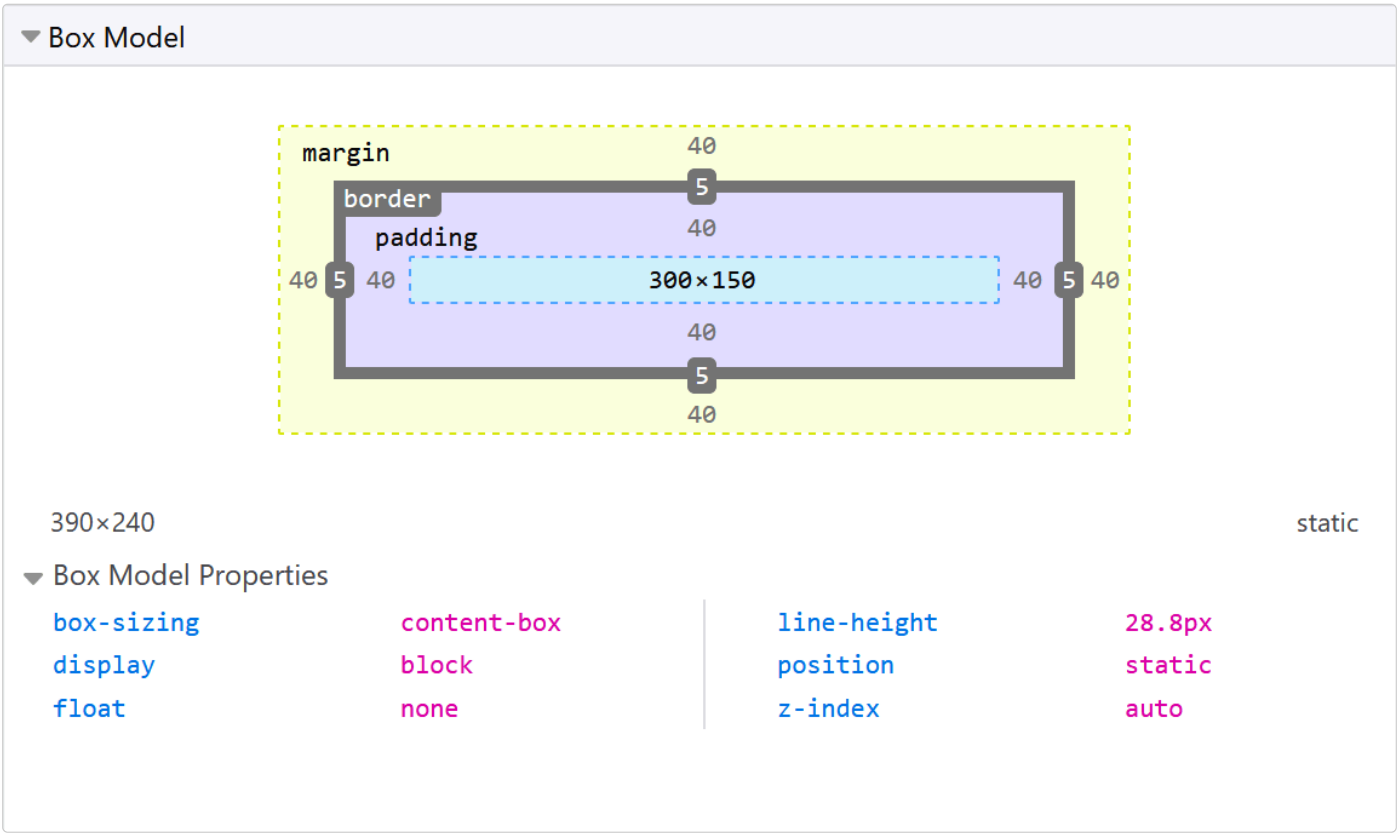
```
<div class="box">I use the standard box model.</div>
<div class="box alternate">I use the alternate box model.</div>
```

Reset

> **Note:** You can find a solution for this task here .

Use browser DevTools to view the box model

Your browser developer tools can make understanding the box model far easier. If you inspect an element in Firefox's DevTools, you can see the size of the element plus its margin, padding, and border. Inspecting an element in this way is a great way to find out if your box is really the size you think it is!



Margins, padding, and borders

You've already seen the `margin`, `padding`, and `border` properties at work in the example above. The properties used in that example are **shorthands** and allow us to set all four sides of the box at once. These shorthands also have equivalent longhand properties, which allow control over the different sides of the box individually.

Let's explore these properties in more detail.

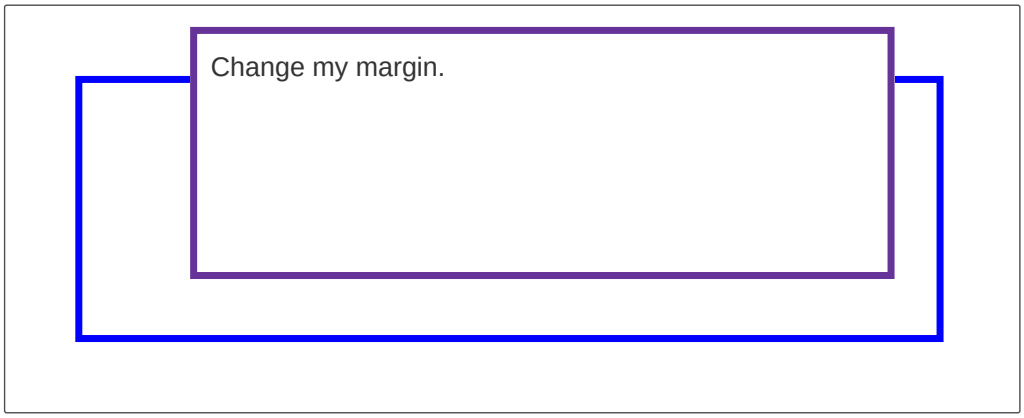
Margin

The margin is an invisible space around your box. It pushes other elements away from the box. Margins can have positive or negative values. Setting a negative margin on one side of your box can cause it to overlap other things on the page. Whether you are using the standard or alternative box model, the margin is always added after the size of the visible box has been calculated.

We can control all margins of an element at once using the `margin` property, or each side individually using the equivalent longhand properties:

- `margin-top`
- `margin-right`
- `margin-bottom`
- `margin-left`

In the example below, try changing the margin values to see how the box is pushed around due to the margin creating or removing space (if it is a negative margin) between this element and the containing element.



Interactive editor

```
.box {
  margin-top: -40px;
  margin-right: 30px;
  margin-bottom: 40px;
  margin-left: 4em;
}
```

```
<div class="container">
  <div class="box">Change my margin.</div>
</div>
```

Reset

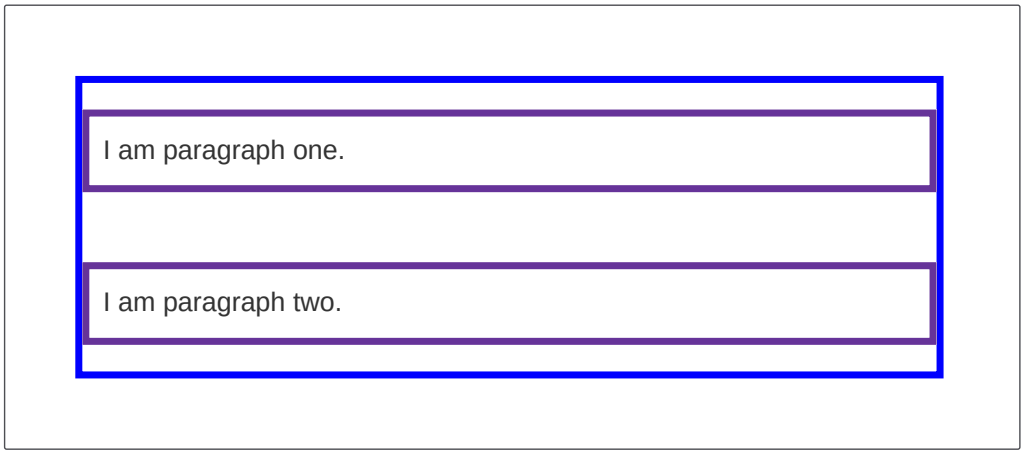
Margin collapsing

Depending on whether two elements whose margins touch have positive or negative margins, the results will be different:

- Two positive margins will combine to become one margin. Its size will be equal to the largest individual margin.
- Two negative margins will collapse and the smallest (furthest from zero) value will be used.
- If one margin is negative, its value will be *subtracted* from the total.

In the example below, we have two paragraphs. The top paragraph has a `margin-bottom` of 50 pixels, the other has a `margin-top` of 30 pixels. The margins have collapsed together so the actual margin between the boxes is 50 pixels and not the total of the two margins.

You can test this by setting the `margin-top` of paragraph two to 0. The visible margin between the two paragraphs will not change — it retains the 50 pixels set in the `margin-bottom` of paragraph one. If you set it to -10px, you'll see that the overall margin becomes 40px — it subtracts from the 50px.



Interactive editor

```
.one {
  margin-bottom: 50px;
}

.two {
  margin-top: 30px;
}
```

```
<div class="container">
  <p class="one">I am paragraph one.</p>
  <p class="two">I am paragraph two.</p>
</div>
```

Reset

A number of rules dictate when margins do and do not collapse. For further information see the detailed page on mastering margin collapsing. The main thing to remember is that margin collapsing is a thing that happens if you are creating space with margins and don't get the space you expect.

Borders

The border is drawn between the margin and the padding of a box. If you are using the standard box model, the size of the border is added to the `width` and `height` of the content box. If you are using the alternative box model, then the bigger the border is, the smaller the content box is, as the border takes up some of that available `width` and `height` of the element box.

For styling borders, there are a large number of properties — there are four borders, and each border has a style, width, and color that we might want to manipulate.

You can set the width, style, or color of all four borders at once using the `border` property.

To set the properties of each side individually, use:

- `border-top`
- `border-right`
- `border-bottom`
- `border-left`

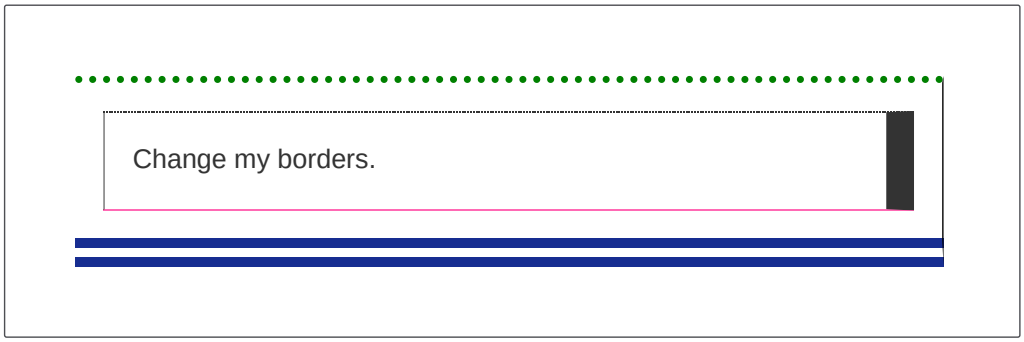
To set the width, style, or color of all sides, use:

- `border-width`
- `border-style`
- `border-color`

To set the width, style, or color of a single side, use one of the more granular longhand properties:

- `border-top-width`
- `border-top-style`
- `border-top-color`
- `border-right-width`
- `border-right-style`
- `border-right-color`
- `border-bottom-width`
- `border-bottom-style`
- `border-bottom-color`
- `border-left-width`
- `border-left-style`
- `border-left-color`

In the example below, we have used various shorthands and longhands to create borders. Play around with the different properties to check that you understand how they work. The MDN pages for the border properties give you information about the different available border styles.



Interactive editor

```
.container {
  border-top: 5px dotted green;
  border-right: 1px solid black;
  border-bottom: 20px double rgb(23 45 145);
}

.box {
  border: 1px solid #333333;
  border-top-style: dotted;
  border-right-width: 20px;
  border-bottom-color: hotpink;
}
```

```
<div class="container">
  <div class="box">Change my borders.</div>
</div>
```

Reset

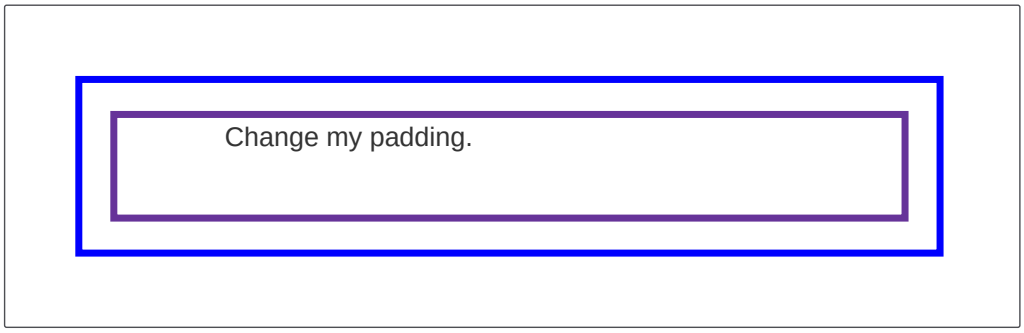
Padding

The padding sits between the border and the content area and is used to push the content away from the border. Unlike margins, you cannot have a negative padding. Any background applied to your element will display behind the padding.

The `padding` property controls the padding on all sides of an element. To control each side individually, use these longhand properties:

- `padding-top`
- `padding-right`
- `padding-bottom`
- `padding-left`

In the example below, you can change the values for padding on the class `.box` to see that this changes where the text begins in relation to the box. You can also change the padding on the class `.container` to create space between the container and the box. You can change the padding on any element to create space between its border and whatever is inside the element.



Interactive editor

```
.box {
  padding-top: 0;
  padding-right: 30px;
  padding-bottom: 40px;
  padding-left: 4em;
}

.container {
  padding: 20px;
}
```

```
<div class="container">
  <div class="box">Change my padding.</div>
</div>
```

Reset

The box model and inline boxes

All of the above fully applies to block boxes. Some of the properties can apply to inline boxes too, such as those created by a `<span>` element.

In the example below, we have a `<span>` inside a paragraph. We have applied a `width`, `height`, `margin`, `border`, and `padding` to it. You can see that the width and height are ignored. The top and bottom margin, padding, and border are respected but don't change the relationship of other content to our inline box. The padding and border overlap other words in the paragraph. The left and right padding, margins, and borders move other content away from the box.

I a raph and this is a span inside that paragraph. A span is an inline element and so does not respect width and height.

Interactive editor

```
span {
  margin: 20px;
  padding: 20px;
  width: 80px;
  height: 50px;
  background-color: lightblue;
  border: 2px solid blue;
}
```

```
<p>
  I am a paragraph and this is a <span>span</span> inside that paragraph. A span
is an inline element and so does not respect width and height.
</p>
```

Reset

Using display: inline-block

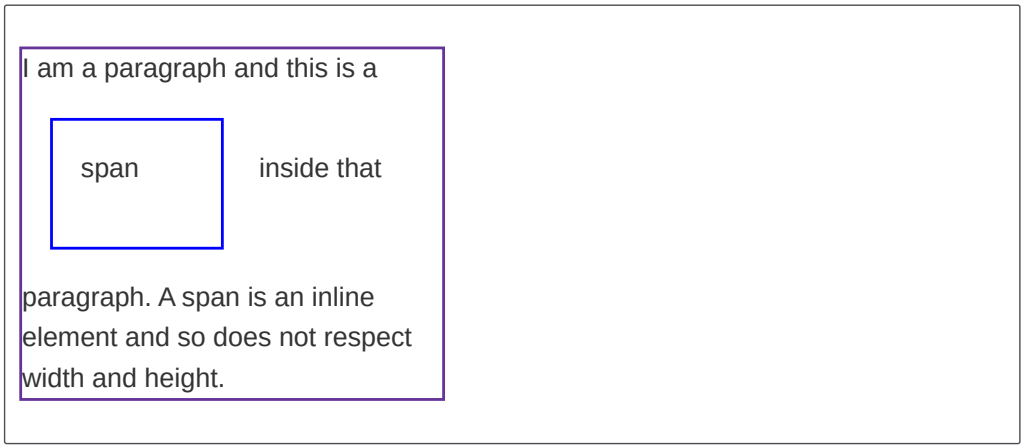
`display: inline-block` is a special value of `display`, which provides a middle ground between `inline` and `block`. Use it if you do not want an item to break onto a new line, but do want it to respect `width` and `height` and avoid the overlapping seen above.

An element with `display: inline-block` does a subset of the block things we already know about:

- The `width` and `height` properties are respected.
- `padding`, `margin`, and `border` will cause other elements to be pushed away from the box.

It does not, however, break onto a new line, and will only become larger than its content if you explicitly add `width` and `height` properties.

In this next example, we have added `display: inline-block` to our `<span>` element. Try changing this to `display: block` or removing the line completely to see the difference in display models.



Interactive editor

```
span {
  margin: 20px;
  padding: 20px;
  width: 80px;
  height: 50px;
  background-color: lightblue;
  border: 2px solid blue;
  display: inline-block;
}
```

```
<p>
  I am a paragraph and this is a <span>span</span> inside that paragraph. A span
is an inline element and so does not respect width and height.
</p>
```

Reset

Where this can be useful is when you want to give a link a larger hit area by adding `padding`. `<a>` is an inline element like `<span>`; you can use `display: inline-block` to allow padding to be set on it, making it easier for a user to click the link.

You see this fairly frequently in navigation bars. The navigation below is displayed in a row using flexbox and we have added padding to the `<a>` element as we want to be able to change the `background-color` when the `<a>` is hovered. The padding appears to overlap the border on the `<ul>` element. This is because the `<a>` is an inline element.

Add `display: inline-block` to the rule with the `.links-list a` selector, and you will see how it fixes this issue by causing the padding to be respected by other elements.

Link one Link two Link three

Interactive editor

```
.links-list a {
  background-color: rgb(179 57 81);
  color: #fff;
  text-decoration: none;
  padding: 1em 2em;
}

.links-list a:hover {
  background-color: rgb(66 28 40);
  color: #fff;
}
```

```
<nav>
  <ul class="links-list">
    <li><a href="">Link one</a></li>
    <li><a href="">Link two</a></li>
    <li><a href="">Link three</a></li>
  </ul>
</nav>
```

Reset

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: The box model.

Summary

That's most of what you need to understand about the box model. You may want to return to this lesson in the future if you ever find yourself confused about how big boxes are in your layout.

In the next article, we'll take a look at how backgrounds and borders can be used to make your plain boxes look more interesting.

CSS values and units

CSS rules contain declarations, which in turn are composed of properties and values. Each property used in CSS has a **value type** that describes what kind of values it is allowed to have. In this lesson, we will take a look at some of the most frequently used value types, what they are, and how they work.

> **Note:** Each CSS property page has a syntax section that lists the value types you can use with that property.

Prerequisites:	Basic software installed, basic knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps).
Objective:	To learn about the different types of values and units used in CSS properties.

What is a CSS value?

In CSS specifications and on the property pages here on MDN you will be able to spot value types as they will be surrounded by angle brackets, such as `<color>` or `<length>`. When you see the value type `<color>` as valid for a particular property, that means you can use any valid color as a value for that property, as listed on the `<color>` reference page.

> **Note:** You'll see CSS value types referred to as *data types*. The terms are basically interchangeable — when you see something in CSS referred to as a data type, it is really just a fancy way of saying value type. The term *value* refers to any particular expression supported by a value type that you choose to use.

> **Note:** CSS value types tend to be enclosed in angle brackets (`<` , `>`) to differentiate them from CSS properties. For example there is a `color` property and a `<color>` data type. This is not to be confused with HTML elements, as they also use angle brackets, but this is something to keep in mind that the context should make clear.

In the following example, we have set the color of our heading using a keyword, and the background using the `rgb()` function:

CSS

```
h1 {
  color: black;
  background-color: rgb(197 93 161);
}
```

A value type in CSS is a way to define a collection of allowable values. This means that if you see `<color>` as valid you don't need to wonder which of the different types of color value can be used — keywords, hex values, `rgb()` functions, etc. You can use *any* available `<color>` values, assuming they are supported by your browser. The page on MDN for each value will give you information about browser support. For example, if you look at the page for `<color>` you will see that the browser compatibility section lists different types of color values and support for them.

Let's have a look at some of the types of values and units you may frequently encounter, with examples so that you can try out different possible values.

Numbers, lengths, and percentages

There are various numeric value types that you might find yourself using in CSS. The following are all classed as numeric:

Data type	Description
`<integer>`	An `<integer>` is a whole number such as `1024` or `-55`.
`<number>`	A `<number>` represents a decimal number — it may or may not have a decimal point with a fractional component. For example, `0.255`, `128`, or `-1.2`.
`<dimension>`	A `<dimension>` is a `<number>` with a unit attached to it. For example, `45deg`, `5s`, or `10px`. `<dimension>` is an umbrella category that includes the `<length>`, `<angle>`, `<time>`, and `<resolution>` types.
`<percentage>`	A `<percentage>` represents a fraction of some other value. For example, `50%`. Percentage values are always relative to another quantity. For example, an element's length is relative to its parent element's length.

Lengths

The numeric type you will come across most frequently is `<length>`. For example, `10px` (pixels) or `30em`. There are two types of lengths used in CSS — relative and absolute. It's important to know the difference in order to understand how big things will become.

Absolute length units

The following are all **absolute** length units — they are not relative to anything else, and are generally considered to always be the same size.

Unit	Name	Equivalent to
`cm`	Centimeters	1cm = 37.8px = 25.2/64in
`mm`	Millimeters	1mm = 1/10th of 1cm
`Q`	Quarter-millimeters	1Q = 1/40th of 1cm
`in`	Inches	1in = 2.54cm = 96px
`pc`	Picas	1pc = 1/6th of 1in
`pt`	Points	1pt = 1/72nd of 1in
`px`	Pixels	1px = 1/96th of 1in

Most of these units are more useful when used for print, rather than screen output. For example, we don't typically use `cm` (centimeters) on screen. The only value that you will commonly use is `px` (pixels).

Relative length units

Relative length units are relative to something else. For example:

- `em` and `rem` are relative to the font size of the parent element and the root element, respectively.
- `vh` and `vw` are relative to the viewport's height and width, respectively.

The benefit of using relative units is that with some careful planning you can make it so the size of text or other elements scales relative to everything else on the page. For a complete list of the relative units available, see the reference page for the `<length>` type.

In this section we'll explore some of the most common relative units.

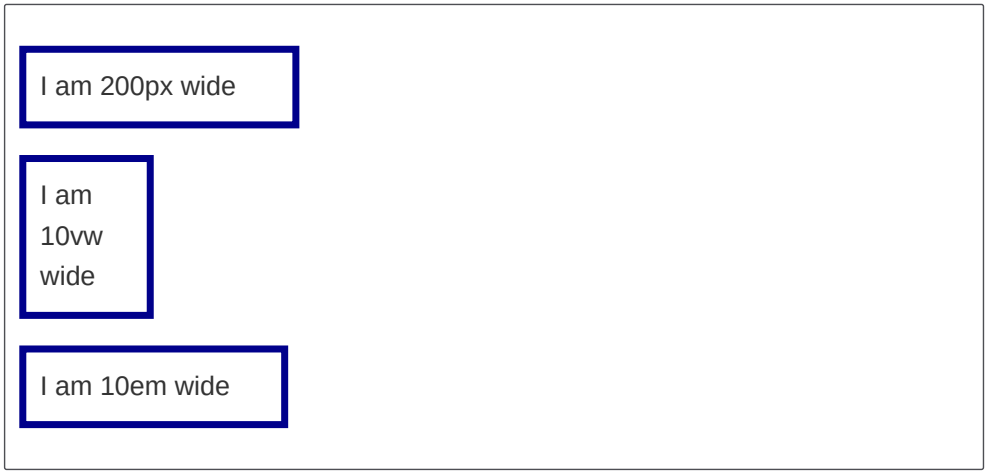
Exploring an example

In the example below, you can see how some relative and absolute length units behave. The first box has a `width` set in pixels. As an absolute unit, this width will remain the same no matter what else changes.

The second box has a width set in `vw` (viewport width) units. This value is relative to the viewport width, and so 10vw is 10 percent of the width of the viewport. If you change the width of your browser window, the size of the box should change. However this example is embedded into the page using an `<iframe>`, so this won't work. To see this in action you'll have to try the example after opening it in its own browser tab .

The third box uses `em` units. These are relative to the font size. I've set a font size of `1em` on the containing `<div>`, which has a class of `.wrapper`. Change this value to `1.5em` and you will see that the font size of all the elements increases, but only the last item will get wider, as its width is relative to that font size.

After following the instructions above, try playing with the values in other ways, to see what you get.



Interactive editor

```
.wrapper {
  font-size: 1em;
}

.px {
  width: 200px;
}

.vw {
  width: 10vw;
}

.em {
  width: 10em;
}
```

```
<div class="wrapper">
  <div class="box px">I am 200px wide</div>
  <div class="box vw">I am 10vw wide</div>
  <div class="box em">I am 10em wide</div>
</div>
```

Reset

ems and rems

`em` and `rem` are the two relative lengths you are likely to encounter most frequently when sizing anything from boxes to text. It's worth understanding how these work, and the differences between them, especially when you start getting on to more complex subjects like styling text or CSS layout. The below example provides a demonstration.

The HTML illustrated below is a set of nested lists — we have two lists in total and both examples have the same HTML. The only difference is that the first has a class of *ems* and the second a class of *rems*.

To start with, we set 16px as the font size on the `<html>` element.

To recap, the em unit means "my parent element's font-size" in the case of typography. The `<li>` elements inside the `<ul>` with a `class` of `ems` take their sizing from their parent. So each successive level of nesting gets progressively larger, as each has its font size set to `1.3em` — 1.3 times its parent's font size.

To recap, the rem unit means "The root element's font-size" (rem stands for "root em"). The `<li>` elements inside the `<ul>` with a `class` of `rems` take their sizing from the root element (`<html>`). This means that each successive level of nesting does not keep getting larger.

However, if you change the `<html>` element's `font-size` in the CSS you will see that everything else changes relative to it — both `rem` - and `em` -sized text.

- One
- Two
- Three
  - Three A
  - Three B
    - Three B 2

- One
- Two
- Three
  - Three A
  - Three B
    - Three B 2

Interactive editor

```
html {
  font-size: 16px;
}

.ems li {
  font-size: 1.3em;
}

.rems li {
  font-size: 1.3rem;
}
```

```
<ul class="ems">
  <li>One</li>
  <li>Two</li>
  <li>Three
    <ul>
      <li>Three A</li>
```

Reset

Line height units

`lh` and `rlh` are relative lengths units similar to `em` and `rem`. The difference between `lh` and `rlh` is that the first one is relative to the line height of the element itself, while the second one is relative to the line height of the root element, usually `<html>`.

Using these units, we can precisely align box decoration to the text. In this example, we use `lh` unit to create notepad-like lines using `repeating-linear-gradient()`. It doesn't matter what's the line height of the text, the lines will always start in the right place.

CSS Play

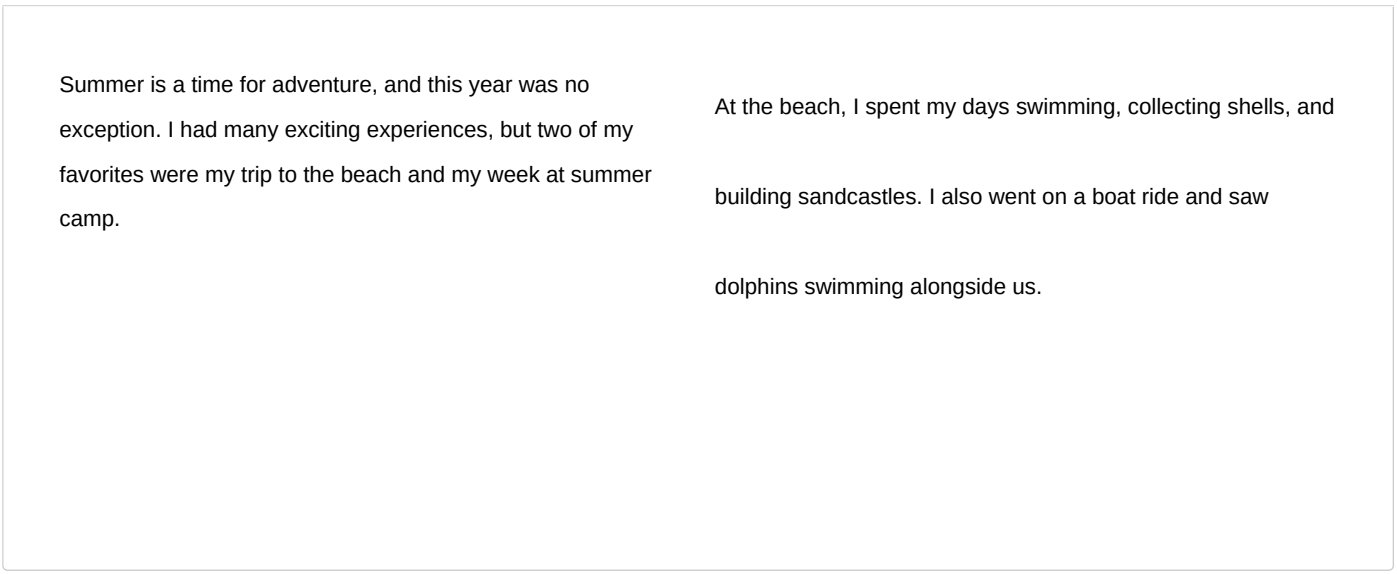
```
p {
  margin: 0;
  background-image: repeating-linear-gradient(
```

```
    to top,
    lightskyblue 0 2px,
    transparent 2px 1lh
  );
}
```

HTML Play

```
<p style="line-height: 2em">
  Summer is a time for adventure, and this year was no exception. I had many
  exciting experiences, but two of my favorites were my trip to the beach and my
  week at summer camp.
</p>

<p style="line-height: 4em">
  At the beach, I spent my days swimming, collecting shells, and building
  sandcastles. I also went on a boat ride and saw dolphins swimming alongside
  us.
</p>
```

Play



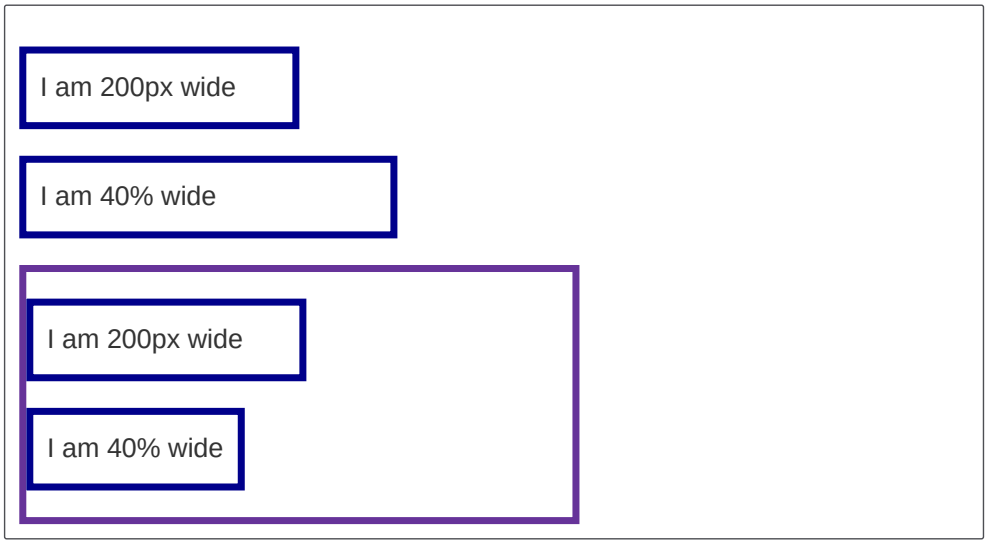
Percentages

In a lot of cases, a percentage is treated in the same way as a length. The thing with percentages is that they are always set relative to some other value. For example, if you set an element's `font-size` as a percentage, it will be a percentage of the `font-size` of the element's parent. If you use a percentage for a `width` value, it will be a percentage of the `width` of the parent.

In the below example the two percentage-sized boxes and the two pixel-sized boxes have the same class names. The sets are 40% and 200px wide respectively.

The difference is that the second set of two boxes is inside a wrapper that is 400 pixels wide. The second 200px wide box is the same width as the first one, but the second 40% box is now 40% of 400px — a lot narrower than the first one!

Try changing the width of the wrapper or the percentage value to see how this works.



Interactive editor

```
.wrapper {
  width: 400px;
  border: 5px solid rebeccapurple;
}

.px {
  width: 200px;
}

.percent {
  width: 40%;
}
```

```
<div class="box px">I am 200px wide</div>
<div class="box percent">I am 40% wide</div>
<div class="wrapper">
  <div class="box px">I am 200px wide</div>
  <div class="box percent">I am 40% wide</div>
</div>
```

Reset

The next example has font sizes set in percentages. Each `<li>` has a `font-size` of 80%; therefore, the nested list items become progressively smaller as they inherit their sizing from their parent.

- One
- Two
- Three
  - Three A
  - Three B
    - Three B 2

Interactive editor

```
li {
  font-size: 80%;
}
```

```
<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three
    <ul>
      <li>Three A</li>
      <li>Three B
        <ul>
          <li>Three B 2</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

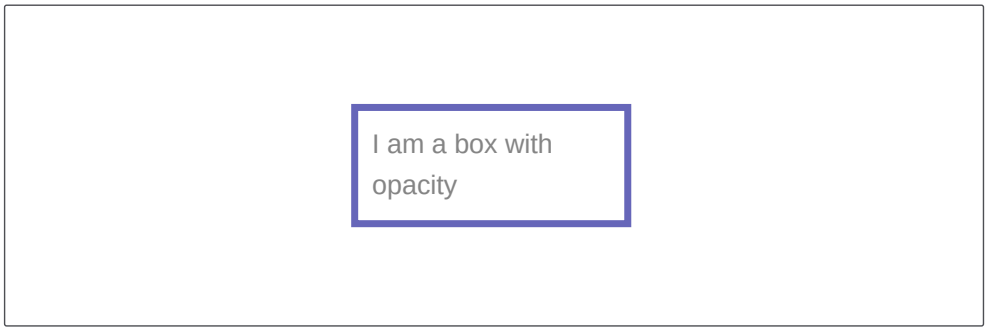
Reset

Note that, while many value types accept a length or a percentage, there are some that only accept length. You can see which values are accepted on the MDN property reference pages. If the allowed value includes `<length-percentage>` then you can use a length or a percentage. If the allowed value only includes `<length>`, it is not possible to use a percentage.

Numbers

Some value types accept numbers, without any unit added to them. An example of a property which accepts a unitless number is the `opacity` property, which controls the opacity of an element (how transparent it is). This property accepts a number between `0` (fully transparent) and `1` (fully opaque).

In the below example, try changing the value of `opacity` to various decimal values between `0` and `1` and see how the box and its contents become more or less opaque.



Interactive editor

```
.box {
  opacity: 0.6;
}
```

```
<div class="wrapper">
  <div class="box">I am a box with opacity</div>
</div>
```

Reset

> **Note:** When you use a number in CSS as a value it should not be surrounded in quotes.

Color

Color values can be used in many places in CSS, whether you are specifying the color of text, backgrounds, borders, and lots more. There are many ways to set color in CSS, allowing you to control plenty of exciting properties.

The standard color system available in modern computers supports 24-bit colors, which allows displaying about 16.7 million distinct colors via a combination of different red, green, and blue channels with 256 different values per channel (256 x 256 x 256 = 16,777,216).

In this section, we'll first look at the most commonly seen ways of specifying colors: using keywords, hexadecimal, and `rgb()` values. We'll also take a quick look at additional color functions, enabling you to recognize them when you see them or experiment with different ways of applying color.

You will likely decide on a color palette and then use those colors — and your favorite way of specifying color — throughout your project. You can mix and match color models, but it's usually best if your entire project uses the same method of declaring colors for consistency!

Color keywords

You will see the color keywords (or 'named colors') used in many MDN code examples. As the `<named-color>` s data type contains a very finite number of color values, these are not commonly used on production websites. As the keyword represents the color as a human-readable text value, named colors are used in code examples to clearly tell the user what color is expected so the learner can focus on the content being taught.

Try playing with different color values in the live examples below, to get more of an idea how they work.

antiquewhite

blueviolet

greenyellow

Interactive editor

```
.one {
  background-color: antiquewhite;
}

.two {
  background-color: blueviolet;
}

.three {
  background-color: greenyellow;
}
```

```
<div class="wrapper">
  <div class="box one">antiquewhite</div>
  <div class="box two">blueviolet</div>
  <div class="box three">greenyellow</div>
</div>
```

Reset

Hexadecimal RGB values

The next type of color value you are likely to encounter is hexadecimal codes. Hexadecimal uses 16 characters from `0-9` and `a-f`, so the entire range is `0123456789abcdef`. Each hex color value consists of a hash/pound symbol (`#`) followed by three or six hexadecimal characters (`#fcc` or `#ffc0cb`, for example), with an optional one or two hexadecimal characters representing the alpha-transparency of the previous three or six character color values.

When using hexadecimal to describe RGB values, each **pair** of hexadecimal characters is a decimal number representing one of the channels — red, green and blue — and allows us to specify any of the 256 available values for each (16 x 16 = 256). These values are less intuitive than keywords for defining colors, but they are a lot more versatile because you can represent any RGB color with them.

#02798b

#c55da1

#128a7d

Interactive editor

```
.one {
  background-color: #02798b;
}

.two {
  background-color: #c55da1;
}

.three {
  background-color: #128a7d;
}
```

```
<div class="wrapper">
  <div class="box one">#02798b</div>
  <div class="box two">#c55da1</div>
  <div class="box three">#128a7d</div>
</div>
```

Reset

Again, try changing the values to see how the colors vary.

RGB values

To create RGB values directly, the `rgb()` function takes three parameters representing **red**, **green**, and **blue** channel values of the colors, with an optional fourth value separated by a slash ('/') representing opacity, in much the same way as hex values. The difference with RGB is that each channel is represented not by two hex digits, but by a decimal number between 0 and 255 or a percent between 0% and 100% inclusive (but not a mixture of the two).

Let's rewrite our last example to use RGB colors:

rgb(2 121 139)

rgb(197 93 161)

rgb(18 138 125)

Interactive editor

```
.one {
  background-color: rgb(2 121 139);
}

.two {
  background-color: rgb(197 93 161);
}

.three {
  background-color: rgb(18 138 125);
}
```

```
<div class="wrapper">
  <div class="box one">rgb(2 121 139)</div>
  <div class="box two">rgb(197 93 161)</div>
  <div class="box three">rgb(18 138 125)</div>
</div>
```

Reset

You can pass a fourth parameter to `rgb()`, which represents the alpha channel of the color, which controls opacity. If you set this value to `0` it will make the color fully transparent, whereas `1` will make it fully opaque. Values in between give you different levels of transparency.

> **Note:** Setting an alpha channel on a color has one key difference to using the `opacity` property we looked at earlier. When you use opacity you make the element and everything inside it opaque, whereas using RGB with an alpha parameter colors only makes the color you are specifying opaque.

In the example below, we have added a background image to the containing block of our colored boxes. We have then set the boxes to have different opacity values — notice how the background shows through more when the alpha channel value is smaller.

rgb(2 121 139 / .3)

rgb(197 93 161 / .7)

rgb(18 138 125 / .9)

Interactive editor

```
.one {
  background-color: rgb(2 121 139 / .3);
}

.two {
  background-color: rgb(197 93 161 / .7);
}

.three {
  background-color: rgb(18 138 125 / .9);
}
```

```
<div class="wrapper">
  <div class="box one">rgb(2 121 139 / .3)</div>
  <div class="box two">rgb(197 93 161 / .7)</div>
  <div class="box three">rgb(18 138 125 / .9)</div>
</div>
```

Reset

In this example, try changing the alpha channel values to see how it affects the color output.

> **Note:** In older versions of CSS, the `rgb()` syntax didn't support an alpha parameter - you needed to use a different function called `rgba()` for that. These days you can pass an alpha parameter to `rgb()`. The `rgba()` function is an alias for `rgb()`.

SRGB values

The `sRGB` color space defines colors in the **red** (r), **green** (g), and **blue** (b) color space.

Using hues to specify a color

If you want to go beyond keywords, hexadecimal, and `rgb()` for colors, you might want to try using `<hue>`. Hue is the property that allows us to tell the difference or similarity between colors like red, orange, yellow, green, blue, etc. The key concept is that you can specify a hue in an `<angle>` because most of the color models describe hues using a color wheel.

There are several color functions that include a `<hue>` component, including `hsl()`, `hwb()`, and `lch()`. Other color functions, like `lab()`, define colors based on what humans can see.

If you want to find out more about these functions and color spaces, see the Applying color to HTML elements using CSS guide, the `<color>` reference that lists all the different ways you can use colors in CSS, and the CSS color module that provides an overview of all the color types in CSS and the properties that use color values.

HWB

A great starting point for using hues in CSS is the `hwb()` function which specifies an `srgb()` color. The three parts are:

- **Hue**: The base shade of the color. This takes a `<hue>` value between 0 and 360, representing the angles around a color wheel.
- **Whiteness**: How white is the color? This takes a value from `0%` (no whiteness) to `100%` (full whiteness).
- **Blackness**: How black is the color? This takes a value from 0% (no blackness) to 100% (full blackness).

HSL

Similar to the `hwb()` function is the `hsl()` function which also specifies an `srgb()` color. HSL uses `Hue`, in addition to `Saturation` and `Lightness`:

- **Hue**
- **Saturation**: How saturated is the color? This takes a value from 0–100%, where 0 is no color (it will appear as a shade of grey), and 100% is full color saturation.
- **Lightness**: How light or bright is the color? This takes a value from 0–100%, where 0 is no light (it will appear completely black) and 100% is full light (it will appear completely white).

The `hsl()` color value also has an optional fourth value, separated from the color with a slash (`/`), representing the alpha transparency.

Let's update the RGB example to use HSL colors instead:

hsl(188 97% 28%)

hsl(321 47% 57%)

hsl(174 77% 31%)

Interactive editor

```
.one {
  background-color: hsl(188 97% 28%);
}

.two {
  background-color: hsl(321 47% 57%);
}

.three {
  background-color: hsl(174 77% 31%);
}
```

```
<div class="wrapper">
  <div class="box one">hsl(188 97% 28%)</div>
  <div class="box two">hsl(321 47% 57%)</div>
  <div class="box three">hsl(174 77% 31%)</div>
</div>
```

Reset

Just like with `rgb()` you can pass an alpha parameter to `hsl()` to specify opacity:

hsl(188 97% 28% / .3)

hsl(321 47% 57% / .7)

hsl(174 77% 31% / .9)

Interactive editor

```
.one {
  background-color: hsl(188 97% 28% / .3);
}

.two {
  background-color: hsl(321 47% 57% / .7);
}

.three {
  background-color: hsl(174 77% 31% / .9);
}
```

```
<div class="wrapper">
  <div class="box one">hsl(188 97% 28% / .3)</div>
  <div class="box two">hsl(321 47% 57% / .7)</div>
  <div class="box three">hsl(174 77% 31% / .9)</div>
</div>
```

Reset

> **Note:** In older versions of CSS, the `hsl()` syntax didn't support an alpha parameter - you needed to use a different function called `hsla()` for that. These days you can pass an alpha parameter to `hsl()`, but for backwards compatibility with old websites, the `hsla()` syntax is still supported, and has exactly the same behavior as `hsl()`.

Images

The `<image>` value type is used wherever an image is a valid value. This can be an actual image file pointed to via a `url()` function, or a gradient.

In the example below, we have demonstrated an image and a gradient in use as a value for the CSS `background-image` property.

Interactive editor

```
.image {
  background-image: url(star.png);
}

.gradient {
  background-image: linear-gradient(90deg, rgb(119 0 255 / 39%), rgb(0 212 255
/ 100%));
}
```

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

Reset

> **Note:** There are some other possible values for `<image>`, however these are newer and currently have poor browser support. Check out the page on MDN for the `<image>` data type if you want to read about them.

Position

The `<position>` value type represents a set of 2D coordinates, used to position an item such as a background image (via `background-position`). It can take keywords such as `top`, `left`, `bottom`, `right`, and `center` to align items with specific bounds of a 2D box, along with lengths, which represent offsets from the top and left-hand edges of the box.

A typical position value consists of two values — the first sets the position horizontally, the second vertically. If you only specify values for one axis the other will default to `center`.

In the following example we have positioned a background image 40px from the top and to the right of the container using a keyword.

Interactive editor

```
.box {
  height: 300px;
  width: 400px;
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: right 40px;
}
```

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

Reset

Play around with these values to see how you can push the image around.

Strings and identifiers

Throughout the examples above, we've seen places where keywords are used as a value (for example `<color>` keywords like `red`, `black`, `rebeccapurple`, and `goldenrod`). These keywords are more accurately described as *identifiers*, a special value that CSS understands. As such they are not quoted — they are not treated as strings.

There are places where you use strings in CSS. For example, when specifying generated content. In this case, the value is quoted to demonstrate that it is a string. In the example below, we use unquoted color keywords along with a quoted generated content string.

This is a string. I know because it is quoted in the CSS.

Interactive editor

```
.box {
  width:400px;
  padding: 1em;
  border-radius: .5em;
  border: 5px solid rebeccapurple;
  background-color: lightblue;
}

.box::after {
  content: "This is a string. I know because it is quoted in the CSS."
}
```

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

Reset

Functions

In programming, a function is a piece of code that does a specific task. Functions are useful because you can write code once, then reuse it many times instead of writing the same logic over and over. Most programming languages not only support functions but also come with convenient built-in functions for common tasks so you don't have to write them yourself from scratch.

CSS also has functions, which work in a similar way to functions in other languages. In fact, we've already seen CSS functions in the Color section above with `rgb()` and `hsl()` functions.

Aside from applying colors, you can use functions in CSS to do a lot of other things. For example Transform functions are a common way to move, rotate, and scale elements on a page. You might see `translate()` for moving something horizontally or vertically, `rotate()` to rotate something, or `scale()` to make something bigger or smaller.

Math functions

When you are creating styles for a project, you will probably start off with numbers like `300px` for lengths or `200ms` for durations. If you want to have these values change based on other values, you will need to do some math. You could calculate the percentage of a value or add a number to another number, then update your CSS with the result.

CSS has support for Math functions, which allow us to perform calculations instead of relying on static values or doing the math in JavaScript. One of the most common math functions is `calc()` which lets you do operations like addition, subtraction, multiplication, and division.

For example, let's say we want to set the width of an element to be 20% of its parent container plus 100px. We can't specify this width with a static value — if the parent uses a percentage width (or a relative unit like `em` or `rem`) then it will vary depending on the context it is used in, and other factors such as the user's device or browser window width. However, we can use `calc()` to set the width of the element to be 20% of its parent container plus 100px. The 20% is based on the width of the parent container (`.wrapper`) and if that width changes, the calculation will change too:

My width is calculated.

Interactive editor

```
.wrapper {
  width: 400px;
}

.box {
  width: calc(20% + 100px);
}
```

```
<div class="wrapper">
  <div class="box">My width is calculated.</div>
</div>
```

Reset

There are many other math functions that you can use in CSS, such as `min()`, `max()`, and `clamp()`; respectively these let you pick the smallest, largest, or middle value from a set of values. You can also use Trigonometric functions like `sin()`, `cos()`, and `tan()` to calculate angles for rotating elements around a point, or choose colors that take a hue angle as a parameter. Exponential functions might also be used for animations and transitions, when you require very specific control over how something moves and looks.

Knowing about CSS functions is useful so you recognize them when you see them. You should start experimenting with them in your projects — they will help you avoid writing custom or repetitive code to achieve results that you can get with regular CSS.

Test your skills!

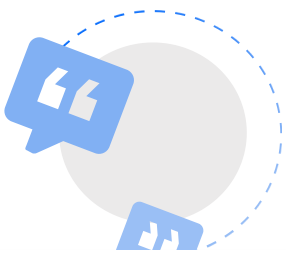
You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Values and units.

Summary

This has been a quick run-through of the most common types of values and units you might encounter. You can have a look at all of the different types on the CSS Values and units reference page — you will encounter many of these in use as you work through these lessons.

The key thing to remember is that each property has a defined list of allowed value types, and each value type has a definition explaining what the values are. You can then look up the specifics here on MDN. For example, understanding that `<image>` also allows you to create a color gradient is useful but perhaps non-obvious knowledge to have!

In the next article, we'll take a look at how items are sized in CSS.



CSS layout

At this point, we've looked at CSS fundamentals, how to style text, and how to style and manipulate the boxes that your content sits inside. Now it's time to look at how to correctly arrange your boxes in relation to the viewport as well as to one another. We've covered the necessary prerequisites, so let's dive deep into CSS layout, looking at such various features as: different display settings, positioning, modern layout tools like flexbox and CSS grid, and some of the legacy techniques you might still want to know about.

Prerequisites

Before starting this module, you should already:

1. Have basic familiarity with HTML, as discussed in the Introduction to HTML module.
2. Be comfortable with CSS fundamentals, as discussed in Introduction to CSS.
3. Understand how to style boxes.

> **Note:** If you are working on a computer/tablet/other device where you don't have the ability to create your own files, you could try out (most of) the code examples in an online coding program such as JSBin or Glitch .

Guides

These articles will provide instruction on the fundamental layout tools and techniques available in CSS. At the end of the lessons is an assessment to help you check your understanding of layout methods by laying out a webpage.

Introduction to CSS layout

This article will recap some of the CSS layout features we've already touched upon in previous modules — such as different `display` values — and introduce some of the concepts we'll be covering throughout this module.

Normal flow

Elements on webpages lay themselves out according to *normal flow* - until we do something to change that. This article explains the basics of normal flow as a grounding for learning how to change it.

Flexbox

Flexbox is a one-dimensional layout method for laying out items in rows or columns. Items flex to fill additional space and shrink to fit into smaller spaces. This article explains all the fundamentals. After studying this guide you can test your flexbox skills to check your understanding before moving on.

Grids

CSS Grid Layout is a two-dimensional layout system for the web. It lets you lay content out in rows and columns, and has many features that make building complex layouts straightforward. This article will give you all you need to know to get started with page layout, then test your grid skills before moving on.

Floats

Originally for floating images inside blocks of text, the `float` property became one of the most commonly used tools for creating multiple column layouts on webpages. With the advent of Flexbox and Grid it has now returned to its original purpose, as this

article explains.

[Positioning]

Positioning allows you to take elements out of the normal document layout flow and make them behave differently, for example, by sitting on top of one another, or by always remaining in the same place inside the browser viewport. This article explains the different `position` values and how to use them.

[Multiple-column layout]

The multiple-column layout specification gives you a method of laying content out in columns, as you might see in a newspaper. This article explains how to use this feature.

[Responsive design]

As more diverse screen sizes have appeared on web-enabled devices, the concept of responsive web design (RWD) has appeared: a set of practices that allows web pages to alter their layout and appearance to suit different screen widths, resolutions, etc. It is an idea that changed the way we design for a multi-device web, and in this article we'll help you understand the main techniques you need to know to master it.

[Beginner's guide to media queries]

The **CSS Media Query** gives you a way to apply CSS only when the browser and device environment matches a rule that you specify, for example, "viewport is wider than 480 pixels". Media queries are a key part of responsive web design because they allow you to create different layouts depending on the size of the viewport. They can also be used to detect other features of the environment your site is running on, for example, whether the user is using a touchscreen rather than a mouse. In this lesson, you will first learn about the syntax used in media queries, and then you will use them in an interactive example showing how a simple design might be made responsive.

[Legacy layout methods]

Grid systems are a very common feature used in CSS layouts. Prior to **CSS Grid Layout**, they tended to be implemented using floats or other layout features. You'd first imagine your layout as a set number of columns (e.g., 4, 6, or 12), and then you'd fit your content columns inside these imaginary columns. In this article, we'll explore how these older methods work so that you understand how they were used if you work on an older project.

[Supporting older browsers]

In this module, we recommend using Flexbox and Grid as the main layout methods for your designs. However, there are bound to be visitors to a site you develop in the future who use older browsers, or browsers which do not support the methods you have used. This will always be the case on the web — as new features are developed, different browsers will prioritize different features. This article explains how to use modern web techniques without excluding users of older technology.

Assessments

The following assessment will test your understanding of the CSS layout methods covered in the guides above.

[Fundamental layout comprehension]

An assessment to test your knowledge of different layout methods by laying out a webpage.

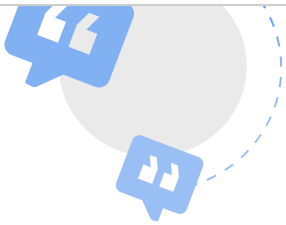
See also

[Practical positioning examples]

This article shows how to build some real-world examples to illustrate what kinds of things you can do with positioning.

CSS layout cookbook

The CSS layout cookbook aims to bring together recipes for common layout patterns, things you might need to implement in your sites. In addition to providing code you can use as a starting point in your projects, these recipes highlight the different ways layout specifications can be used and the choices you can make as a developer.



Introduction to CSS layout

This article will recap some of the CSS layout features we've already touched upon in previous modules, such as different `display` values, as well as introduce some of the concepts we'll be covering throughout this module.

Prerequisites:	The basics of HTML (study Introduction to HTML), and an idea of How CSS works (study Introduction to CSS.)
Objective:	To give you an overview of CSS page layout techniques. Each technique can be learned in greater detail in subsequent tutorials.

CSS page layout techniques allow us to take elements contained in a web page and control where they're positioned relative to the following factors: their default position in normal layout flow, the other elements around them, their parent container, and the main viewport/window. The page layout techniques we'll be covering in more detail in this module are:

- Normal flow
- The `display` property
- Flexbox
- Grid
- Floats
- Positioning
- Table layout
- Multiple-column layout

Each technique has its uses, advantages, and disadvantages. No technique is designed to be used in isolation. By understanding what each layout method is designed for you'll be in a good position to understand which method is most appropriate for each task.

Normal flow

Normal flow is how the browser lays out HTML pages by default when you do nothing to control page layout. Let's look at a quick HTML example:

HTML Play

```
<p>I love my cat.</p>

<ul>
  <li>Buy cat food</li>
  <li>Exercise</li>
  <li>Cheer up friend</li>
</ul>

<p>The end!</p>
```

By default, the browser will display this code as follows:

Play

Note how the HTML is displayed in the exact order in which it appears in the source code, with elements stacked on top of one another — the first paragraph, followed by the unordered list, followed by the second paragraph.

The elements that appear one below the other are described as **block** elements, in contrast to **inline** elements, which appear beside one another like the individual words in a paragraph.

> **Note:** The direction in which block element contents are laid out is described as the Block Direction. The Block Direction runs vertically in a language such as English, which has a horizontal writing mode. It would run horizontally in any language with a Vertical Writing Mode, such as Japanese. The corresponding Inline Direction is the direction in which inline contents (such as a sentence) would run.

For many of the elements on your page, the normal flow will create exactly the layout you need. However, for more complex layouts you will need to alter this default behavior using some of the tools available to you in CSS. Starting with a well-structured HTML document is very important because you can then work with the way things are laid out by default rather than fighting against it.

The methods that can change how elements are laid out in CSS are:

- **The `display` property** — Standard values such as `block`, `inline` or `inline-block` can change how elements behave in normal flow, for example, by making a block-level element behave like an inline-level element (see Types of CSS boxes for more information). We also have entire layout methods that are enabled via specific `display` values, for example, CSS Grid and Flexbox, which alter how child elements are laid out inside their parents.
- **Floats** — Applying a `float` value such as `left` can cause block-level elements to wrap along one side of an element, like the way images sometimes have text floating around them in magazine layouts.
- **The `position` property** — Allows you to precisely control the placement of boxes inside other boxes. `static` positioning is the default in normal flow, but you can cause elements to be laid out differently using other values, for example, as fixed to the top of the browser viewport.
- **Table layout** — Features designed for styling parts of an HTML table can be used on non-table elements using `display: table` and associated properties.
- **Multi-column layout** — The Multi-column layout properties can cause the content of a block to lay out in columns, as you might see in a newspaper.

The display property

The main methods for achieving page layout in CSS all involve specifying values for the `display` property. This property allows us to change the default way something displays. Everything in normal flow has a default value for `display`; i.e., a default way that elements are set to behave. For example, the fact that paragraphs in English display one below the other is because they are styled with `display: block`. If you create a link around some text inside a paragraph, that link remains inline with the rest of the text, and doesn't break onto a new line. This is because the `<a>` element is `display: inline` by default.

You can change this default display behavior. For example, the `<li>` element is `display: block` by default, meaning that list items display one below the other in our English document. If we were to change the display value to `inline` they would display next to each other, as words would do in a sentence. The fact that you can change the value of `display` for any element means that you can pick HTML elements for their semantic meaning without being concerned about how they will look. The way they look is something that you can change.

In addition to being able to change the default presentation by turning an item from `block` to `inline` and vice versa, there are some more involved layout methods that start out as a value of `display`. However, when using these you will generally need to invoke additional properties. The two values most important for our discussion of layout are `display: flex` and `display: grid`.

Flexbox

Flexbox is the short name for the Flexible Box Layout CSS module, designed to make it easy for us to lay things out in one dimension — either as a row or as a column. To use flexbox, you apply `display: flex` to the parent element of the elements you want to lay out; all its direct children then become *flex items*. We can see this in a simple example.

Setting display: flex

The HTML markup below gives us a containing element with a class of `wrapper`, inside of which are three `<div>` elements. By default these would display as block elements, that is, below one another in our English language document.

However, if we add `display: flex` to the parent, the three items now arrange themselves into columns. This is due to them becoming *flex items* and being affected by some initial values that flexbox sets on the flex container. They are displayed in a row because the property `flex-direction` of the parent element has an initial value of `row`. They all appear to stretch in height because the property `align-items` of their parent element has an initial value of `stretch`. This means that the items stretch to the height of the flex container, which in this case is defined by the tallest item. The items all line up at the start of the container, leaving any extra space at the end of the row.

CSS Play

```
.wrapper {
  display: flex;
}
```

HTML Play

```
<div class="wrapper">
  <div class="box1">One</div>
  <div class="box2">Two</div>
  <div class="box3">Three</div>
</div>
```

Play

Setting the flex property

In addition to properties that can be applied to a *flex container*, there are also properties that can be applied to *flex items*. These properties, among other things, can change the way that items *flex*, enabling them to expand or contract according to available space.

As a simple example, we can add the `flex` property to all of our child items, and give it a value of `1`. This will cause all of the items to grow and fill the container, rather than leaving space at the end. If there is more space then the items will become wider; if there is less space they will become narrower. In addition, if you add another element to the markup, the other items will all become smaller to make space for it; the items all together continue taking up all the space.

CSS Play

```
.wrapper {
  display: flex;
}

.wrapper > div {
  flex: 1;
}
```

HTML Play

```
<div class="wrapper">
  <div class="box1">One</div>
  <div class="box2">Two</div>
  <div class="box3">Three</div>
</div>
```

Play

> **Note:** This has been a very short introduction to what is possible in Flexbox. To find out more, see our Flexbox article.

Grid Layout

While flexbox is designed for one-dimensional layout, Grid Layout is designed for two dimensions — lining things up in rows and columns.

Setting display: grid

Similar to flexbox, we enable Grid Layout with its specific display value — `display: grid`. The below example uses similar markup to the flex example, with a container and some child elements. In addition to using `display: grid`, we also define some row and column *tracks* for the parent using the `grid-template-rows` and `grid-template-columns` properties respectively. We've defined three columns, each of `1fr`, as well as two rows of `100px`. We don't need to put any rules on the child elements; they're automatically placed into the cells our grid's created.

CSS Play

```
.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 100px 100px;
  gap: 10px;
}
```

HTML Play

```
<div class="wrapper">
  <div class="box1">One</div>
  <div class="box2">Two</div>
  <div class="box3">Three</div>
  <div class="box4">Four</div>
  <div class="box5">Five</div>
  <div class="box6">Six</div>
</div>
```

Play

Placing items on the grid

Once you have a grid, you can explicitly place your items on it, rather than relying on the auto-placement behavior seen above. In the next example below, we've defined the same grid, but this time with three child items. We've set the start and end line of each item using the `grid-column` and `grid-row` properties. This causes the items to span multiple tracks.

CSS Play

```
.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 100px 100px;
  gap: 10px;
}

.box1 {
  grid-column: 2 / 4;
  grid-row: 1;
}

.box2 {
  grid-column: 1;
  grid-row: 1 / 3;
}

.box3 {
```

```
  grid-row: 2;
  grid-column: 3;
}
```

HTML Play

```
<div class="wrapper">
  <div class="box1">One</div>
  <div class="box2">Two</div>
  <div class="box3">Three</div>
</div>
```

Play

> **Note:** These two examples reveal just a small sample of the power of Grid layout. To learn more, see our Grid Layout article.

The rest of this guide covers other layout methods that are less important for the main layout of your page, but still help to achieve specific tasks. By understanding the nature of each layout task you will soon find that when you look at a particular component of your design, the type of layout most suitable for it will often be clear.

Floats

Floating an element changes the behavior of that element and the block level elements that follow it in normal flow. The floated element is moved to the left or right and removed from normal flow, and the surrounding content *floats* around it.

The `float` property has four possible values:

- `left` — Floats the element to the left.
- `right` — Floats the element to the right.
- `none` — Specifies no floating at all. This is the default value.
- `inherit` — Specifies that the value of the `float` property should be inherited from the element's parent element.

In the example below, we float a `<div>` left and give it a `margin` on the right to push the surrounding text away from it. This gives us the effect of text wrapped around the boxed element, and is most of what you need to know about floats as used in modern web design.

HTML Play

```
<h1>Simple float example</h1>

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

<p>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus aliquam
  dolor, eu lacinia lorem placerat vulputate. Duis felis orci, pulvinar id metus
  ut, rutrum luctus orci. Cras porttitor imperdiet nunc, at ultricies tellus
  laoreet sit amet. Sed auctor cursus massa at porta. Integer ligula ipsum,
  tristique sit amet orci vel, viverra egestas ligula. Curabitur vehicula tellus
  neque, ac ornare ex malesuada et. In vitae convallis lacus. Aliquam erat
  volutpat. Suspendisse ac imperdiet turpis. Aenean finibus sollicitudin eros
  pharetra congue. Duis ornare egestas augue ut luctus. Proin blandit quam nec
  lacus varius commodo et a urna. Ut id ornare felis, eget fermentum sapien.
</p>
```

CSS Play

```
.box {
  float: left;
  width: 150px;
  height: 150px;
  margin-right: 30px;
}
```

Play

> **Note:** Floats are fully explained in our lesson on the float and clear properties. Prior to techniques such as Flexbox and Grid Layout, floats were used as a method of creating column layouts. You may still come across these methods on the web; we will cover these in the lesson on legacy layout methods.

Positioning techniques

Positioning allows you to move an element from where it would otherwise be placed in normal flow over to another location. Positioning isn't a method for creating the main layouts of a page; it's more about managing and fine-tuning the position of specific items on a page.

There are, however, useful techniques for obtaining specific layout patterns that rely on the `position` property. Understanding positioning also helps in understanding normal flow, and what it means to move an item out of the normal flow.

There are five types of positioning you should know about:

- **Static positioning** is the default that every element gets. It just means "put the element into its normal position in the document layout flow — nothing special to see here".
- **Relative positioning** allows you to modify an element's position on the page, moving it relative to its position in normal flow, as well as making it overlap other elements on the page.
- **Absolute positioning** moves an element completely out of the page's normal layout flow, like it's sitting on its own separate layer. From there, you can fix it to a position relative to the edges of its closest positioned ancestor (which becomes `<html>` if no other ancestors are positioned). This is useful for creating complex layout effects, such as tabbed boxes where different content panels sit on top of one another and are shown and hidden as desired, or information panels that sit off-screen by default, but can be made to slide on screen using a control button.
- **Fixed positioning** is very similar to absolute positioning except that it fixes an element relative to the browser viewport, not another element. This is useful for creating effects such as a persistent navigation menu that always stays in the same place on the screen as the rest of the content scrolls.
- **Sticky positioning** is a newer positioning method that makes an element act like `position: relative` until it hits a defined offset from the viewport, at which point it acts like `position: fixed`.

Simple positioning example

To provide familiarity with these page layout techniques, we'll show you a couple of quick examples. Our examples will all feature the same HTML structure (a heading followed by three paragraphs), which is as follows:

HTML Play

```
<h1>Positioning</h1>

<p>I am a basic block level element.</p>
<p class="positioned">I am a basic block level element.</p>
<p>I am a basic block level element.</p>
```

This HTML will be styled by default using the following CSS:

CSS Play

```
body {
  width: 500px;
  margin: 0 auto;
}

p {
  background-color: rgb(207 232 220);
  border: 2px solid rgb(79 185 227);
  padding: 10px;
  margin: 10px;
  border-radius: 5px;
}

.positioned {
  background: rgb(255 84 104 / 30%);
```

```
  border: 2px solid rgb(255 84 104);
}
```

The rendered output is as follows:

Play

Relative positioning

Relative positioning allows you to offset an item from its default position in normal flow. This means you could achieve a task such as moving an icon down a bit so it lines up with a text label. To do this, we could add the following rule to add relative positioning:

CSS Play

```
.positioned {
  position: relative;
  top: 30px;
  left: 30px;
}
```

Here we give our middle paragraph a `position` value of `relative`. This doesn't do anything on its own, so we also add `top` and `left` properties. These serve to move the affected element down and to the right. This might seem like the opposite of what you were expecting, but you need to think of it as the element being pushed on its left and top sides, which results in it moving right and down.

Adding this code will give the following result:

Play

Absolute positioning

Absolute positioning is used to completely remove an element from the normal flow and instead position it using offsets from the edges of a containing block.

Going back to our original non-positioned example, we could add the following CSS rule to implement absolute positioning:

CSS Play

```css
.positioned {
  position: absolute;
  top: 30px;
  left: 30px;
}
```

Here we give our middle paragraph a `position` value of `absolute` and the same `top` and `left` properties as before. Adding this code will produce the following result:

Play

This is very different! The positioned element has now been completely separated from the rest of the page layout and sits over the top of it. The other two paragraphs now sit together as if their positioned sibling doesn't exist. The `top` and `left` properties have a different effect on absolutely positioned elements than they do on relatively positioned elements. In this case, the offsets have been calculated from the top and left of the page. It is possible to change the parent element that becomes this container and we will take a look at that in the lesson on positioning.

Fixed positioning

Fixed positioning removes our element from document flow in the same way as absolute positioning. However, instead of the offsets being applied from the container, they are applied from the viewport. Because the item remains fixed in relation to the viewport, we can create effects such as a menu that remains fixed as the page scrolls beneath it.

For this example, our HTML contains three paragraphs of text so that we can scroll through the page, as well as a box with the property of `position: fixed`.

HTML Play

```html
<h1>Fixed positioning</h1>

<div class="positioned">Fixed</div>

<p>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus aliquam
  dolor, eu lacinia lorem placerat vulputate. Duis felis orci, pulvinar id metus
  ut, rutrum luctus orci.
</p>
```

```
<p>
  Cras porttitor imperdiet nunc, at ultricies tellus laoreet sit amet. Sed
  auctor cursus massa at porta. Integer ligula ipsum, tristique sit amet orci
  vel, viverra egestas ligula. Curabitur vehicula tellus neque, ac ornare ex
  malesuada et.
</p>

<p>
  In vitae convallis lacus. Aliquam erat volutpat. Suspendisse ac imperdiet
  turpis. Aenean finibus sollicitudin eros pharetra congue. Duis ornare egestas
  augue ut luctus. Proin blandit quam nec lacus varius commodo et a urna. Ut id
  ornare felis, eget fermentum sapien.
</p>
```

CSS Play

```
.positioned {
  position: fixed;
  top: 30px;
  left: 30px;
}
```

Play

Sticky positioning

Sticky positioning is the final positioning method that we have at our disposal. It mixes relative positioning with fixed positioning. When an item has `position: sticky`, it'll scroll in normal flow until it hits offsets from the viewport that we have defined. At that point, it becomes "stuck" as if it had `position: fixed` applied.

CSS Play

```
.positioned {
  position: sticky;
  top: 30px;
  left: 30px;
}
```

Play

> **Note:** To find out more about positioning, see our Positioning article.

Table layout

HTML tables are fine for displaying tabular data, but many years ago — before even basic CSS was supported reliably across browsers — web developers used to also use tables for entire web page layouts, putting their headers, footers, columns, etc. into various table rows and columns. This worked at the time, but it has many problems: table layouts are inflexible, very heavy on markup, difficult to debug, and semantically wrong (e.g., screen reader users have problems navigating table layouts).

The way that a table looks on a webpage when you use table markup is due to a set of CSS properties that define table layout. These same properties can also be used to lay out elements that aren't tables, a use which is sometimes described as "using CSS tables".

The example below shows one such use. It must be noted, that using CSS tables for layout should be considered a legacy method at this point, and should only be used to support old browsers that lack support for Flexbox or Grid.

Let's look at an example. First, some simple markup that creates an HTML form. Each input element has a label, and we've also included a caption inside a paragraph. Each label/input pair is wrapped in a `<div>` for layout purposes.

HTML Play

```
<form>
  <p>First of all, tell us your name and age.</p>
  <div>
    <label for="fname">First name:</label>
    <input type="text" id="fname" />
  </div>
  <div>
    <label for="lname">Last name:</label>
    <input type="text" id="lname" />
  </div>
  <div>
    <label for="age">Age:</label>
    <input type="text" id="age" />
  </div>
</form>
```

As for the CSS, most of it's fairly ordinary except for the uses of the `display` property. The `<form>`, `<div>`s, and `<label>`s and `<input>`s have been told to display like a table, table rows, and table cells respectively. Basically, they'll act like HTML table markup, causing the labels and inputs to line up nicely by default. All we then have to do is add a bit of sizing, margin, etc., to make everything look a bit nicer and we're done.

You'll notice that the caption paragraph has been given `display: table-caption;`, which makes it act like a table `<caption>`, and `caption-side: bottom;` to tell the caption to sit on the bottom of the table for styling purposes, even though the markup is before the `<input>` elements in the source. This allows for a nice bit of flexibility.

CSS Play

```
html {
  font-family: sans-serif;
}

form {
  display: table;
  margin: 0 auto;
}
```

```
form div {
  display: table-row;
}

form label,
form input {
  display: table-cell;
  margin-bottom: 10px;
}

form label {
  width: 200px;
  padding-right: 5%;
  text-align: right;
}

form input {
  width: 300px;
}

form p {
  display: table-caption;
  caption-side: bottom;
  width: 300px;
  color: #999;
  font-style: italic;
}
```

This gives us the following result:

Play

You can also see this example live at css-tables-example.html (see the source code too.)

> **Note:** Table layout, unlike the other topics of this page, won't be further covered in this module due to its legacy application.

Multi-column layout

The multi-column layout CSS module provides us a way to lay out content in columns, similar to how text flows in a newspaper. While reading up and down columns is less useful in a web context due to the users having to scroll up and down, arranging content into columns can, nevertheless, be a useful technique.

To turn a block into a multi-column container, we use either the `column-count` property, which tells the browser *how many* columns we would like to have, or the `column-width` property, which tells the browser to fill the container with as many columns as possible of a *specified width*.

In the below example, we start with a block of HTML inside a containing `<div>` element with a class of `container`.

HTML Play

```
<div class="container">
  <h1>Multi-column Layout</h1>

  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
    aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
    pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc, at
    ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
  </p>

  <p>
    Nam vulputate diam nec tempor bibendum. Donec luctus augue eget malesuada
    ultrices. Phasellus turpis est, posuere sit amet dapibus ut, facilisis sed
    est. Nam id risus quis ante semper consectetur eget aliquam lorem.
  </p>

  <p>
    Vivamus tristique elit dolor, sed pretium metus suscipit vel. Mauris
    ultricies lectus sed lobortis finibus. Vivamus eu urna eget velit cursus
    viverra quis vestibulum sem. Aliquam tincidunt eget purus in interdum. Cum
    sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus
    mus.
  </p>
</div>
```

We're using a `column-width` of 200 pixels on that container, causing the browser to create as many 200 pixel columns as will fit. Whatever space is left between the columns will be shared.

CSS Play

```
.container {
  column-width: 200px;
}
```

Play

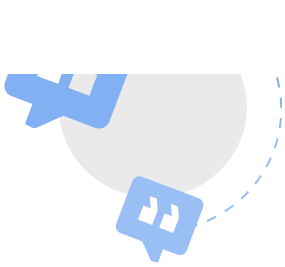
Summary

This article has provided a brief summary of all the layout technologies you should know about. Read on for more information on each individual technology!

Was this page helpful to you?

Yes No

Learn how to contribute.



This page was last modified on Dec 15, 2023 by MDN contributors.

Normal Flow

This article explains normal flow, or the way that webpage elements lay themselves out if you haven't changed their layout.

Prerequisites:	The basics of HTML (study Introduction to HTML), and an idea of How CSS works (study Introduction to CSS.)
Objective:	To explain how browsers layout web pages by default, before we begin to make changes.

As detailed in the last lesson introducing layout, elements on a webpage lay out in normal flow if you haven't applied any CSS to change the way they behave. And, as we began to discover, you can change how elements behave either by adjusting their position in normal flow or by removing them from it altogether. Starting with a solid, well-structured document that's readable in normal flow is the best way to begin any webpage. It ensures that your content is readable even if the user's using a very limited browser or a device such as a screen reader that reads out the content of the page. In addition, since normal flow is designed to make a readable document, by starting in this way you're working *with* the document rather than struggling *against* it as you make changes to the layout.

Before digging deeper into different layout methods, it's worth revisiting some of the things you have studied in previous modules with regard to normal document flow.

How are elements laid out by default?

The process begins as the boxes of individual elements are laid out in such a way that any padding, border, or margin they happen to have is added to their content. This is what we call the **box model**.

By default, a block-level element's content fills the available inline space of the parent element containing it, growing along the block dimension to accommodate its content. The size of inline-level elements is just the size of their content. You can set `width` or `height` on some elements that have a default `display` property value of `inline`, like `<img>`, but `display` value will still remain `inline`.

If you want to control the `display` property of an inline-level element in this manner, use CSS to set it to behave like a block-level element (e.g., with `display: block;` or `display: inline-block;`, which mixes characteristics from both).

That explains how elements are structured individually, but how about the way they're structured when they interact with one another? The normal layout flow (mentioned in the layout introduction article) is the system by which elements are placed inside the browser's viewport. By default, block-level elements are laid out in the *block flow direction*, which is based on the parent's writing mode (*initial*: horizontal-tb). Each element will appear on a new line below the last one, with each one separated by whatever margin that's been specified. In English, for example, (or any other horizontal, top to bottom writing mode) block-level elements are laid out vertically.

Inline elements behave differently. They don't appear on new lines; instead, they all sit on the same line along with any adjacent (or wrapped) text content as long as there is space for them to do so inside the width of the parent block level element. If there isn't space, then the overflowing content will move down to a new line.

If two vertically adjacent elements both have a margin set on them and their margins touch, the larger of the two margins remains and the smaller one disappears. This is known as **margin collapsing**. Collapsing margins is only relevant in the **vertical direction**.

Let's look at a simple example that explains all of this:

HTML Play

```html
<h1>Basic document flow</h1>

<p>
  I am a basic block level element. My adjacent block level elements sit on new
  lines below me.
</p>

<p>
  By default we span 100% of the width of our parent element, and we are as tall
  as our child content. Our total width and height is our content + padding +
  border width/height.
</p>

<p>
  We are separated by our margins. Because of margin collapsing, we are
  separated by the width of one of our margins, not both.
</p>

<p>
  Inline elements <span>like this one</span> and <span>this one</span> sit on
  the same line along with adjacent text nodes, if there is space on the same
  line. Overflowing inline elements will
  <span>wrap onto a new line if possible (like this one containing text)</span>,
  or just go on to a new line if not, much like this image will do:
  <img src="long.jpg" alt="snippet of cloth" />
</p>
```

CSS Play

```css
body {
  width: 500px;
  margin: 0 auto;
}

p {
  background: rgb(255 84 104 / 30%);
  border: 2px solid rgb(255 84 104);
  padding: 10px;
  margin: 10px;
}

span {
  background: white;
  border: 1px solid black;
}
```

Play

Summary

In this lesson you've learned the basics of normal flow — the default layout for CSS elements. By understanding how inline elements, block elements, and margins behave by default, it'll be easier to modify their behavior in the future.

In the next article, we'll build on this knowledge by making changes to CSS elements using flexbox.

Flexbox

Flexbox is a one-dimensional layout method for arranging items in rows or columns. Items *flex* (expand) to fill additional space or shrink to fit into smaller spaces. This article explains all the fundamentals.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of how CSS works (study Introduction to CSS.)
Objective:	To learn how to use the Flexbox layout system to create web layouts.

Why Flexbox?

For a long time, the only reliable cross-browser compatible tools available for creating CSS layouts were features like floats and positioning. These work, but in some ways they're also limiting and frustrating.

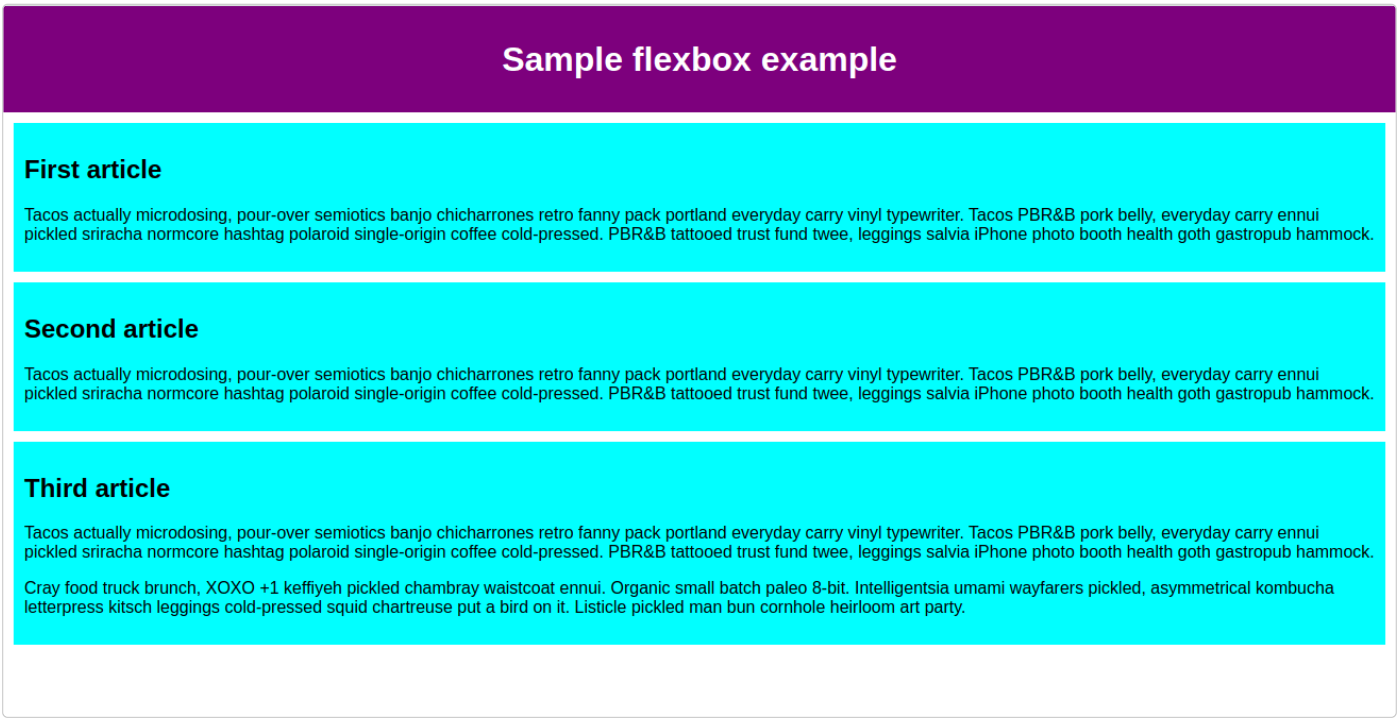
The following simple layout designs are either difficult or impossible to achieve with such tools in any kind of convenient, flexible way:

- Vertically centering a block of content inside its parent.
- Making all the children of a container take up an equal amount of the available width/height, regardless of how much width/height is available.
- Making all columns in a multiple-column layout adopt the same height even if they contain a different amount of content.

As you'll see in subsequent sections, flexbox makes a lot of layout tasks much easier. Let's dig in!

Introducing a simple example

In this article, you'll work through a series of exercises to help you understand how flexbox works. To get started, you should make a local copy of the first starter file — flexbox0.html from our GitHub repo. Load it in a modern browser (like Firefox or Chrome) and have a look at the code in your code editor. You can also see it live here .



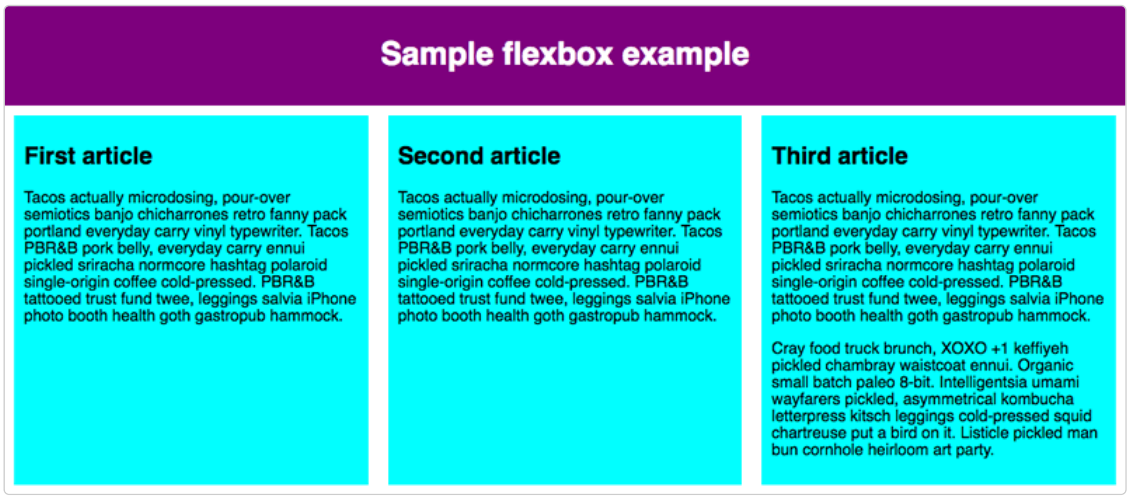
You'll see that we have a `<header>` element with a top level heading inside it and a `<section>` element containing three `<article>`s. We're going to use these to create a fairly standard three column layout.

Specifying what elements to lay out as flexible boxes

To start with, we need to select which elements are to be laid out as flexible boxes. To do this, we set a special value of `display` on the parent element of the elements you want to affect. In this case we want to lay out the `<article>` elements, so we set this on the `<section>`:

```css
section {
  display: flex;
}
```

This causes the `<section>` element to become a **flex container** and its children become **flex items**. This is what it looks like:



This single declaration gives us everything we need. Incredible, right? We have a multiple column layout with equal-sized columns, and the columns are all the same height. This is because the default values given to flex items (the children of the flex container) are set up to solve common problems such as this.

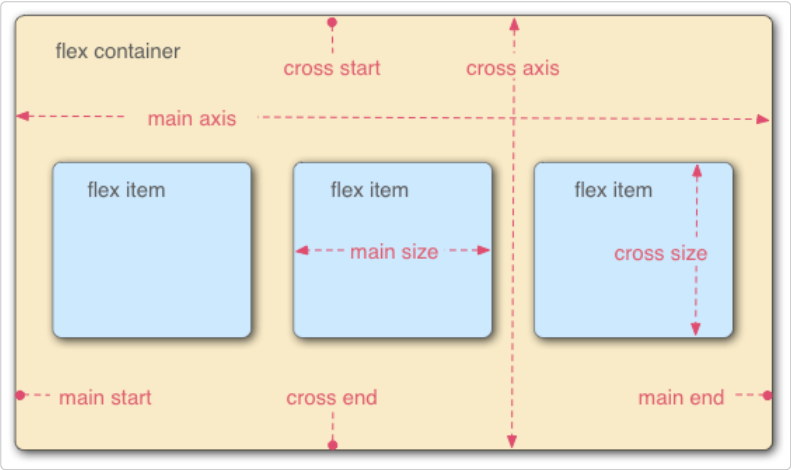
Let's recap what's happening here. Adding a `display` value of `flex` to an element makes it a flex container. The container is displayed as Block-level content in terms of how it interacts with the rest of the page. When the element is converted to a flex container, its children are converted to (and laid out as) flex items.

You can make the container inline using an outside `display` value (e.g., `display: inline flex`), which affects how the container itself is laid out in the page. The legacy `inline-flex` display value displays the container as inline as well. We'll focus on how the contents of the container behave in this tutorial, but if you want to see the effect of inline versus block layout, you can have a look at the value comparison on the `display` property page.

The next sections explain in more detail what flex items are and what happens inside an element when you make it a flex container.

The flex model

When elements are laid out as flex items, they are laid out along two axes:



- The **main axis** is the axis running in the direction the flex items are laid out in (for example, as a row across the page, or a column down the page.) The start and end of this axis are called the **main start** and **main end**.
- The **cross axis** is the axis running perpendicular to the direction the flex items are laid out in. The start and end of this axis are called the **cross start** and **cross end**.
- The parent element that has `display: flex` set on it (the `<section>` in our example) is called the **flex container**.
- The items laid out as flexible boxes inside the flex container are called **flex items** (the `<article>` elements in our example).

Bear this terminology in mind as you go through subsequent sections. You can always refer back to it if you get confused about any of the terms being used.

Columns or rows?

Flexbox provides a property called `flex-direction` that specifies which direction the main axis runs (which direction the flexbox children are laid out in). By default this is set to `row`, which causes them to be laid out in a row in the direction your browser's default language works in (left to right, in the case of an English browser).

Try adding the following declaration to your `<section>` rule:

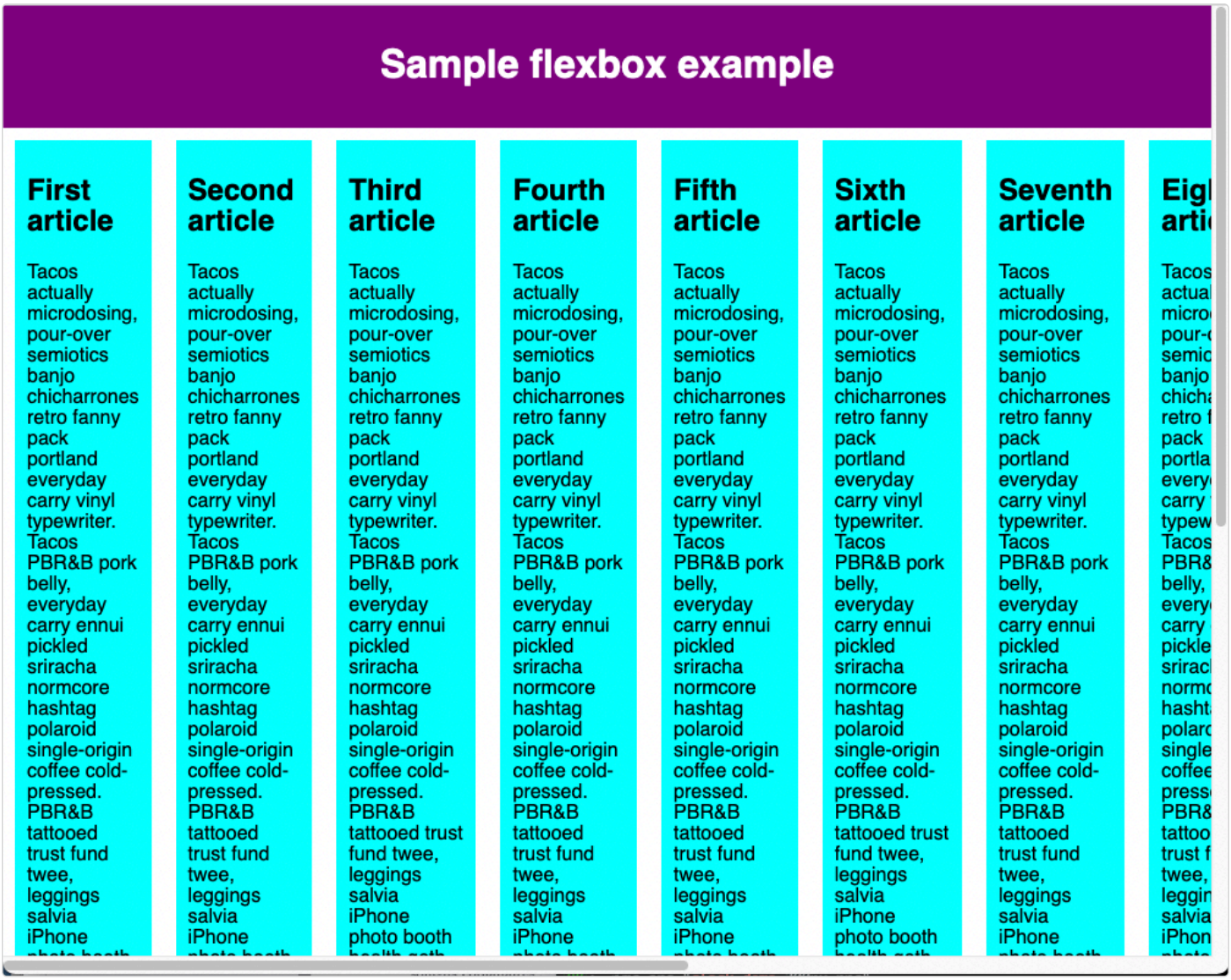
```
CSS
flex-direction: column;
```

You'll see that this puts the items back in a column layout, much like they were before we added any CSS. Before you move on, delete this declaration from your example.

> **Note:** You can also lay out flex items in a reverse direction using the `row-reverse` and `column-reverse` values. Experiment with these values too!

Wrapping

One issue that arises when you have a fixed width or height in your layout is that eventually your flexbox children will overflow their container, breaking the layout. Have a look at our flexbox-wrap0.html example and try viewing it live (take a local copy of this file now if you want to follow along with this example):



Here we see that the children are indeed breaking out of their container. One way in which you can fix this is to add the following declaration to your `<section>` rule:

CSS

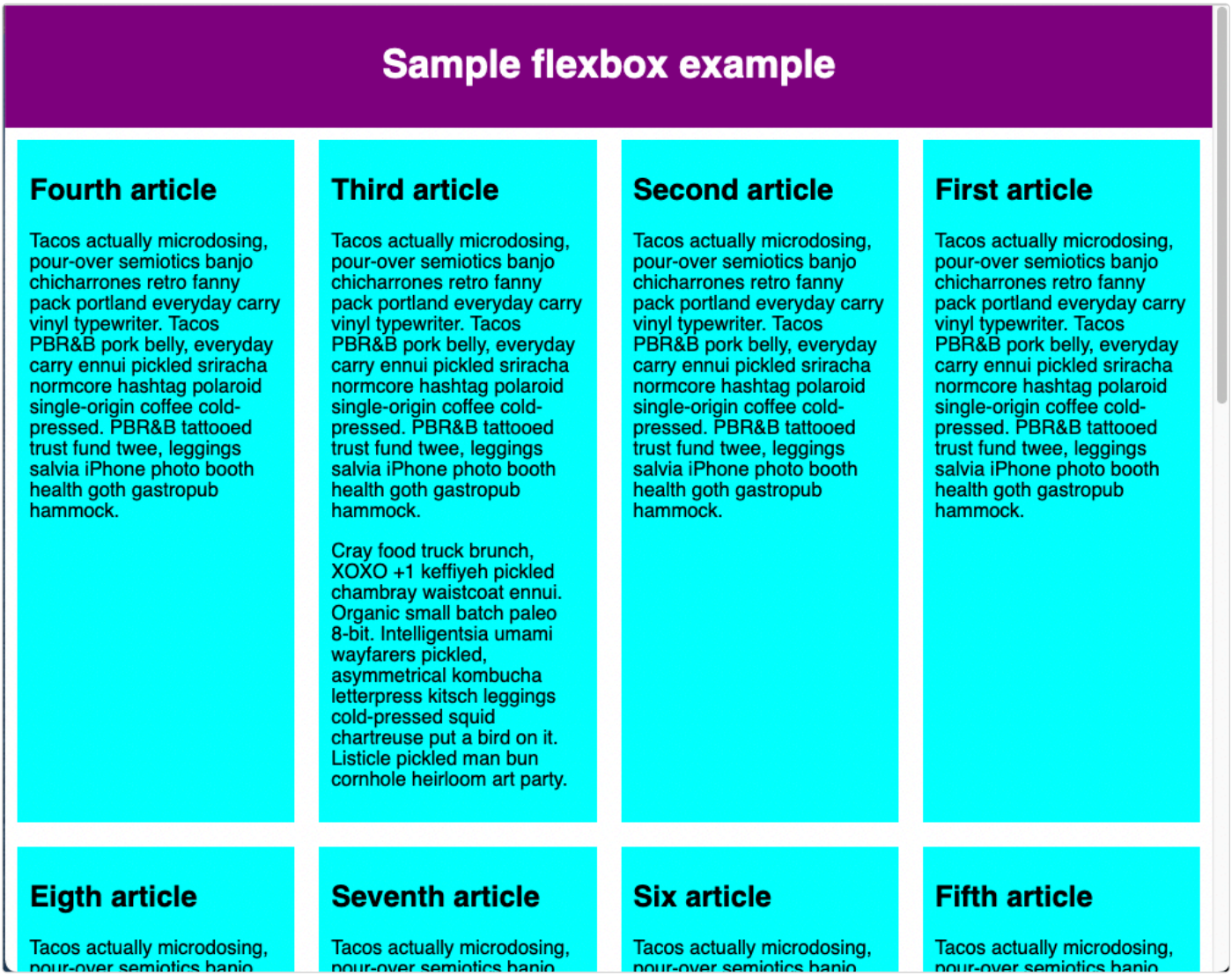
```
flex-wrap: wrap;
```

Also, add the following declaration to your `<article>` rule:

CSS

```
flex: 200px;
```

Try this now. You'll see that the layout looks much better with this included:



We now have multiple rows. Each row has as many flexbox children fitted into it as is sensible. Any overflow is moved down to the next line. The `flex: 200px` declaration set on the articles means that each will be at least 200px wide. We'll discuss this property in more detail later on. You might also notice that the last few children on the last row are each made wider so that the entire row is still filled.

But there's more we can do here. First of all, try changing your `flex-direction` property value to `row-reverse`. Now you'll see that you still have your multiple row layout, but it starts from the opposite corner of the browser window and flows in reverse.

flex-flow shorthand

At this point it's worth noting that a shorthand exists for `flex-direction` and `flex-wrap`: `flex-flow`. So, for example, you can replace

CSS

```
flex-direction: row;
flex-wrap: wrap;
```

with

CSS

```
flex-flow: row wrap;
```

Flexible sizing of flex items

Let's now return to our first example and look at how we can control what proportion of space flex items take up compared to the other flex items. Fire up your local copy of flexbox0.html , or take a copy of flexbox1.html as a new starting point (see it live).

First, add the following rule to the bottom of your CSS:

CSS

```
article {
  flex: 1;
}
```

This is a unitless proportion value that dictates how much available space along the main axis each flex item will take up compared to other flex items. In this case, we're giving each `<article>` element the same value (a value of 1), which means they'll all take up an equal amount of the spare space left after properties like padding and margin have been set. This value is proportionally shared among the flex items: giving each flex item a value of 400000 would have exactly the same effect.

Now add the following rule below the previous one:

CSS

```
article:nth-of-type(3) {
  flex: 2;
}
```

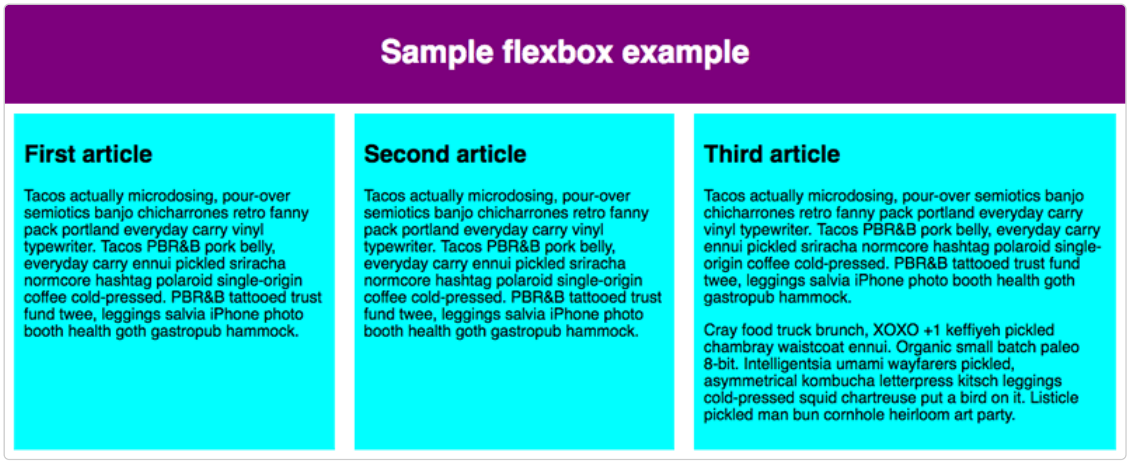
Now when you refresh, you'll see that the third `<article>` takes up twice as much of the available width as the other two. There are now four proportion units available in total (since 1 + 1 + 2 = 4). The first two flex items have one unit each, so they each take 1/4 of the available space. The third one has two units, so it takes up 2/4 of the available space (or one-half).

You can also specify a minimum size value within the flex value. Try updating your existing article rules like so:

CSS

```
article {
  flex: 1 200px;
}

article:nth-of-type(3) {
  flex: 2 200px;
}
```

This basically states, "Each flex item will first be given 200px of the available space. After that, the rest of the available space will be shared according to the proportion units." Try refreshing and you'll see a difference in how the space is shared.



The real value of flexbox can be seen in its flexibility/responsiveness. If you resize the browser window or add another `<article>` element, the layout continues to work just fine.

flex: shorthand versus longhand

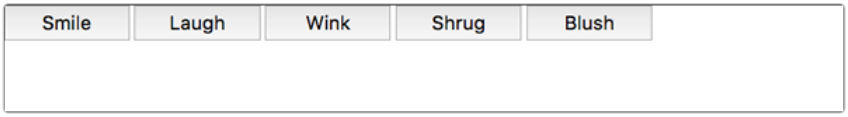
`flex` is a shorthand property that can specify up to three different values:

- The unitless proportion value we discussed above. This can be specified separately using the `flex-grow` longhand property.
- A second unitless proportion value, `flex-shrink`, which comes into play when the flex items are overflowing their container. This value specifies how much an item will shrink in order to prevent overflow. This is quite an advanced flexbox feature and we won't be covering it any further in this article.
- The minimum size value we discussed above. This can be specified separately using the `flex-basis` longhand value.

We'd advise against using the longhand flex properties unless you really have to (for example, to override something previously set). They lead to a lot of extra code being written, and they can be somewhat confusing.

Horizontal and vertical alignment

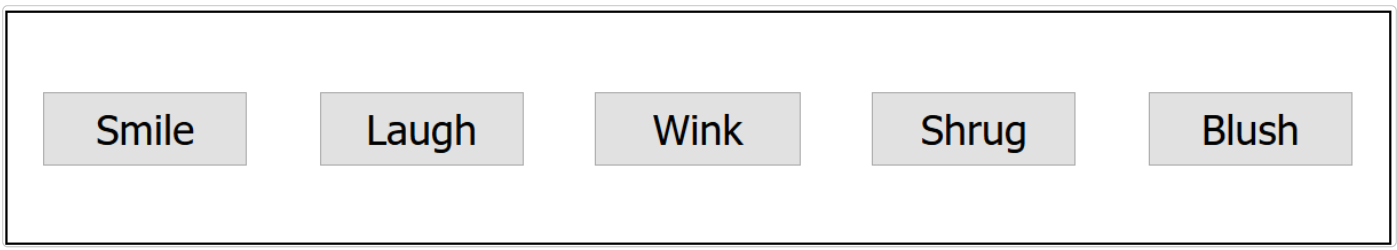
You can also use flexbox features to align flex items along the main or cross axis. Let's explore this by looking at a new example: flex-align0.html (see it live also). We're going to turn this into a neat, flexible button/toolbar. At the moment you'll see a horizontal menu bar with some buttons jammed into the top left-hand corner.



First, take a local copy of this example.

Now, add the following to the bottom of the example's CSS:

```css
div {
  display: flex;
  align-items: center;
  justify-content: space-around;
}
```



Refresh the page and you'll see that the buttons are now nicely centered horizontally and vertically. We've done this via two new properties.

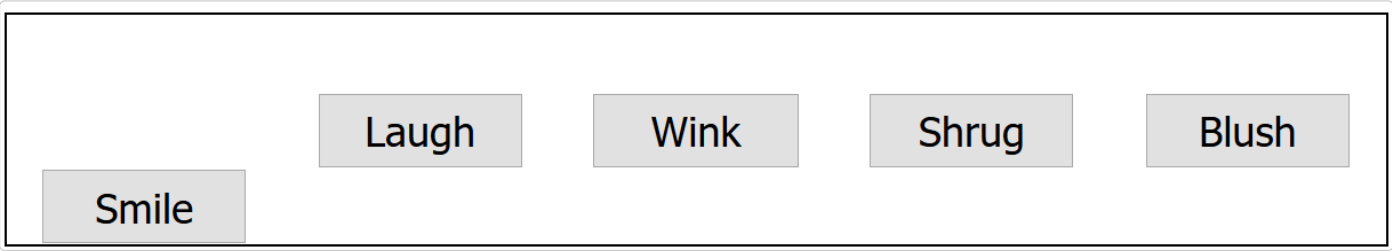
`align-items` controls where the flex items sit on the cross axis.

- By default, the value is `stretch`, which stretches all flex items to fill the parent in the direction of the cross axis. If the parent doesn't have a fixed height in the cross axis direction, then all flex items will become as tall as the tallest flex item. This is how our first example had columns of equal height by default.

- The `center` value that we used in our above code causes the items to maintain their intrinsic dimensions, but be centered along the cross axis. This is why our current example's buttons are centered vertically.
- You can also have values like `flex-start` and `flex-end`, which will align all items at the start and end of the cross axis respectively. See `align-items` for the full details.

You can override the `align-items` behavior for individual flex items by applying the `align-self` property to them. For example, try adding the following to your CSS:

```
CSS
button:first-child {
  align-self: flex-end;
}
```



Have a look at what effect this has and remove it again when you've finished.

`justify-content` controls where the flex items sit on the main axis.

- The default value is `flex-start`, which makes all the items sit at the start of the main axis.
- You can use `flex-end` to make them sit at the end.
- `center` is also a value for `justify-content`. It'll make the flex items sit in the center of the main axis.
- The value we've used above, `space-around`, is useful — it distributes all the items evenly along the main axis with a bit of space left at either end.
- There is another value, `space-between`, which is very similar to `space-around` except that it doesn't leave any space at either end.

The `justify-items` property is ignored in flexbox layouts.

We'd like to encourage you to play with these values to see how they work before you continue.

Ordering flex items

Flexbox also has a feature for changing the layout order of flex items without affecting the source order. This is another thing that is impossible to do with traditional layout methods.

The code for this is simple. Try adding the following CSS to your button bar example code:

```
CSS
button:first-child {
  order: 1;
}
```

Refresh and you'll see that the "Smile" button has moved to the end of the main axis. Let's talk about how this works in a bit more detail:

- By default, all flex items have an `order` value of 0.
- Flex items with higher specified order values will appear later in the display order than items with lower order values.
- Flex items with the same order value will appear in their source order. So if you have four items whose order values have been set as 2, 1, 1, and 0 respectively, their display order would be 4th, 2nd, 3rd, then 1st.
- The 3rd item appears after the 2nd because it has the same order value and is after it in the source order.

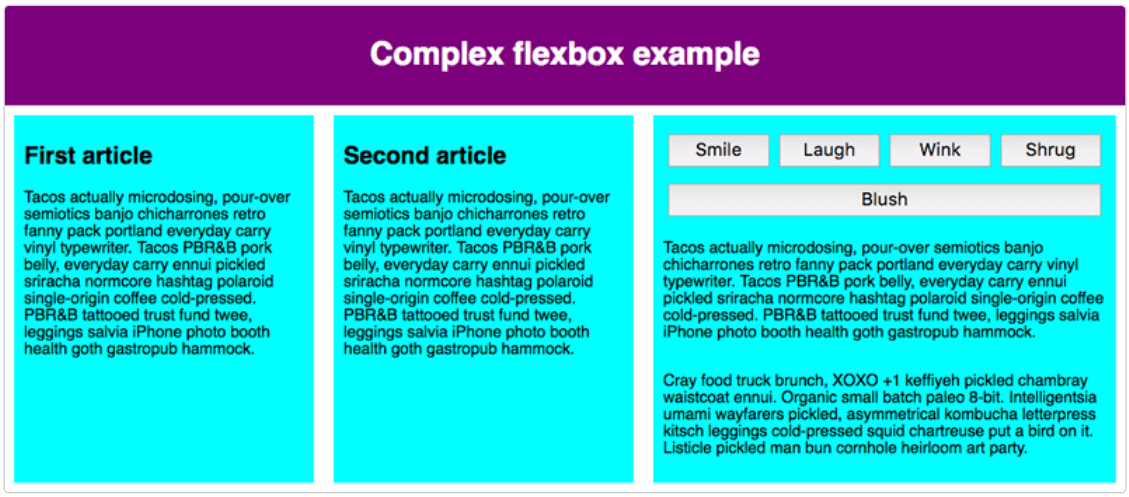
You can set negative order values to make items appear earlier than items whose value is 0. For example, you could make the "Blush" button appear at the start of the main axis using the following rule:

CSS

```
button:last-child {
  order: -1;
}
```

Nested flex boxes

It's possible to create some pretty complex layouts with flexbox. It's perfectly OK to set a flex item to also be a flex container, so that its children are also laid out like flexible boxes. Have a look at complex-flexbox.html (see it live also).



The HTML for this is fairly simple. We've got a `<section>` element containing three `<article>` s. The third `<article>` contains three `<div>` s, and the first `<div>` contains five `<button>` s:

```
section - article
          article
          article - div - button
                    div   button
                    div   button
                          button
                          button
```

Let's look at the code we've used for the layout.

First of all, we set the children of the `<section>` to be laid out as flexible boxes.

CSS

```
section {
  display: flex;
}
```

Next, we set some flex values on the `<article>`s themselves. Take special note of the second rule here: we're setting the third `<article>` to have its children laid out like flex items too, but this time we're laying them out like a column.

CSS

```
article {
  flex: 1 200px;
}

article:nth-of-type(3) {
  flex: 3 200px;
  display: flex;
  flex-flow: column;
}
```

Next, we select the first `<div>`. We first use `flex: 1 100px;` to effectively give it a minimum height of 100px, then we set its children (the `<button>` elements) to also be laid out like flex items. Here we lay them out in a wrapping row and align them in the center of the available space as we did with the individual button example we saw earlier.

CSS

```
article:nth-of-type(3) div:first-child {
  flex: 1 100px;
  display: flex;
  flex-flow: row wrap;
  align-items: center;
  justify-content: space-around;
}
```

Finally, we set some sizing on the button. This time by giving it a flex value of 1 auto. This has a very interesting effect, which you'll see if you try resizing your browser window width. The buttons will take up as much space as they can. As many will fit on a line as is comfortable; beyond that, they'll drop to a new line.

CSS

```
button {
  flex: 1 auto;
  margin: 5px;
  font-size: 18px;
  line-height: 1.5;
}
```

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Flexbox.

Summary

That concludes our tour of the basics of Flexbox. We hope you had fun and will have a good play around with it as you proceed further with your learning. Next, we'll have a look at another important aspect of CSS layouts: CSS Grids.

See also

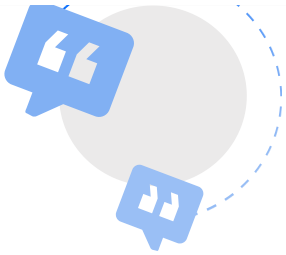
- CSS-Tricks Guide to Flexbox — an article explaining everything about Flexbox in a visually appealing way
- Flexbox Froggy — an educational game to learn and better understand the basics of Flexbox

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.



This page was last modified on Jan 10, 2024 by MDN contributors.

Grids

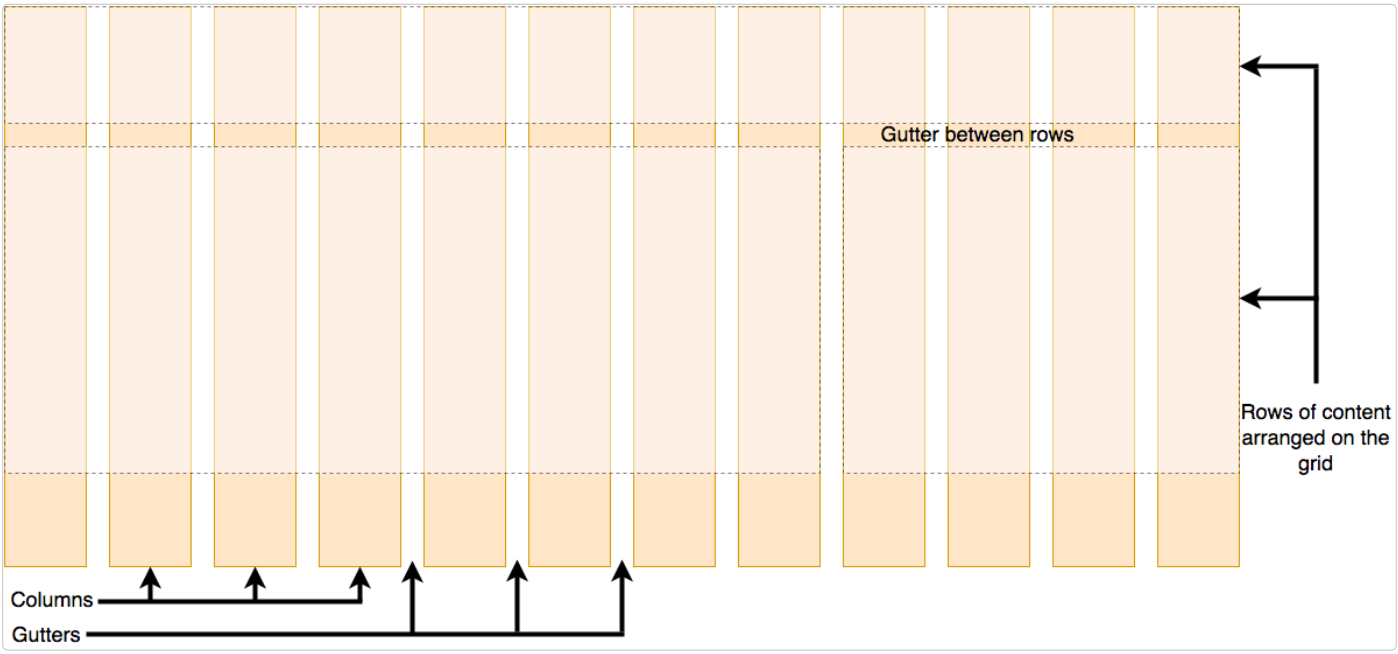
CSS grid layout is a two-dimensional layout system for the web. It lets you organize content into rows and columns and offers many features to simplify the creation of complex layouts. This article will explain all you need to know to get started with grid layout.

Prerequisites:	HTML basics (study Introduction to HTML) and an idea of how CSS works (study Introduction to CSS and Styling boxes.)
Objective:	To understand the fundamental concepts of grid layout as well as how to implement it with CSS Grid.

What is grid layout?

A grid is a collection of horizontal and vertical lines creating a pattern against which we can line up our design elements. They help us to create layouts in which our elements won't jump around or change width as we move from page to page, providing greater consistency on our websites.

A grid will typically have **columns**, **rows**, and then gaps between each row and column. The gaps are commonly referred to as **gutters**.



Creating your grid in CSS

Having decided on the grid that your design needs, you can use CSS Grid Layout to create it. We'll look at the basic features of Grid Layout first and then explore how to create a simple grid system for your project. The following video provides a nice visual explanation of using CSS Grid:

Defining a grid

Let's try out grid layouts with the help of an example. Download and open the starting point file in your text editor and browser (you can also see it live here). You will see an example with a container, which has some child items. By default, these items are displayed in a normal flow, causing them to appear one below the other. For the initial part of this lesson, we'll be using this file to see how its grid behaves.

Similar to how you define flexbox, you define a grid layout by setting the value of the `display` property to `grid` . As in the case of flexbox, the `display: grid` property transforms all the direct children of the container into grid items. Add the following CSS to your file:

CSS Play

```
.container {
  display: grid;
}
```

Unlike Flexbox, the items will not immediately look any different. Declaring `display: grid` gives you a one column grid, so your items will continue to display one below the other as they do in normal flow.

To see something that looks more grid-like, we'll need to add some columns to the grid. Let's add three 200-pixel columns. You can use any length unit or percentage to create these column tracks.

CSS Play

```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px;
}
```

Add the second declaration to your CSS rule, then reload the page. You should see that the items have rearranged themselves such that there's one in each cell of the grid.

Play

Flexible grids with the fr unit

In addition to creating grids using lengths and percentages, we can use `fr`. The `fr` unit represents one fraction of the available space in the grid container to flexibly size grid rows and columns.

Change your track listing to the following definition, creating three `1fr` tracks:

CSS Play

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

You now have flexible tracks. The `fr` unit distributes space proportionally. You can specify different positive values for your tracks like so:

CSS Play

```
.container {
  display: grid;
  grid-template-columns: 2fr 1fr 1fr;
}
```

The first track gets `2fr` of the available space and the other two tracks get `1fr`, making the first track larger. You can mix `fr` units with fixed length units. In this case, the space needed for the fixed tracks is used up first before the remaining space is distributed to the other tracks.

Play

> **Note:** The `fr` unit distributes *available* space, not *all* space. Therefore, if one of your tracks has something large inside it, there will be less free space to share.

Gaps between tracks

To create gaps between tracks, we use the properties:

- `column-gap` for gaps between columns
- `row-gap` for gaps between rows
- `gap` as a shorthand for both

CSS Play

```
.container {
  display: grid;
  grid-template-columns: 2fr 1fr 1fr;
  gap: 20px;
}
```

These gaps can be any length unit or percentage, but not an `fr` unit.

Play

> **Note:** The `gap` properties (`column-gap`, `row-gap` and `gap`) used to be prefixed by `grid-`. The spec has changed but the prefixed versions will be maintained as an alias. To be on the safe side and make your code more bulletproof, you can add both properties:
>
> CSS Play
>
> ```
> .container {
>   display: grid;
>   grid-template-columns: 2fr 1fr 1fr;
>   grid-gap: 20px;
>   gap: 20px;
> }
> ```

Repeating track listings

You can repeat all or merely a section of your track listing using the CSS `repeat()` function. Change your track listing to the following:

CSS

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 20px;
}
```

You'll now get three `1fr` tracks just as before. The first value passed to the `repeat()` function specifies the number of times you want the listing to repeat, while the second value is a track listing, which may be one or more tracks that you want to repeat.

Implicit and explicit grids

Up to this point, we've specified only column tracks, but rows are automatically created to hold the content. This concept highlights the distinction between explicit and implicit grids. Here's a bit more about the difference between the two types of grids:

- **Explicit grid** is created using `grid-template-columns` or `grid-template-rows`.
- **Implicit grid** extends the defined explicit grid when content is placed outside of that grid, such as into the rows by drawing additional grid lines.

By default, tracks created in the implicit grid are `auto` sized, which in general means that they're large enough to contain their content. If you wish to give implicit grid tracks a size, you can use the `grid-auto-rows` and `grid-auto-columns` properties. If you add `grid-auto-rows` with a value of `100px` to your CSS, you'll see that those created rows are now 100 pixels tall.

CSS Play

```css
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-auto-rows: 100px;
  gap: 20px;
}
```

Play

The minmax() function

Our 100-pixel tall tracks won't be very useful if we add content into those tracks that is taller than 100 pixels, in which case it would cause an overflow. It might be better to have tracks that are *at least* 100 pixels tall and can still expand if more content becomes added. A fairly basic fact about the web is that you never really know how tall something is going to be — additional content or larger font sizes can cause problems with designs that attempt to be pixel perfect in every dimension.

The `minmax()` function lets us set a minimum and maximum size for a track, for example, `minmax(100px, auto)`. The minimum size is 100 pixels, but the maximum is `auto`, which will expand to accommodate more content. Try changing `grid-auto-rows` to use a minmax value:

CSS

```css
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
```

```
  grid-auto-rows: minmax(100px, auto);
  gap: 20px;
}
```

If you add extra content, you'll see that the track expands to allow it to fit. Note that the expansion happens right along the row.

As many columns as will fit

We can combine some of the lessons we've learned about track listing, repeat notation, and `minmax()` to create a useful pattern. Sometimes it's helpful to be able to ask grid to create as many columns as will fit into the container. We do this by setting the value of `grid-template-columns` using the `repeat()` function, but instead of passing in a number, pass in the keyword `auto-fit`. For the second parameter of the function we use `minmax()` with a minimum value equal to the minimum track size that we would like to have and a maximum of `1fr`.

Try this in your file now using the CSS below:

CSS Play

```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
  grid-auto-rows: minmax(100px, auto);
  gap: 20px;
}
```

Play

This works because grid is creating as many 200-pixel columns as will fit into the container, then sharing whatever space is leftover among all the columns. The maximum is `1fr` which, as we already know, distributes space evenly between tracks.

Line-based placement

We now move on from creating a grid to placing things on the grid. Our grid always has lines — these are numbered beginning with 1 and relate to the writing mode of the document. For example, column line 1 in English (written left-to-right) would be on the left-hand side of the grid and row line 1 at the top, while in Arabic (written right-to-left), column line 1 would be on the right-hand side.

To position items along these lines, we can specify the start and end lines of the grid area where an item should be placed. There are four properties we can use to do this:

- `grid-column-start`
- `grid-column-end`
- `grid-row-start`
- `grid-row-end`

These properties accept line numbers as their values, so we can specify that an item should start on line 1 and end on line 3, for example. Alternatively, you can also use shorthand properties that let you specify the start and end lines simultaneously, separated by a forward slash `/`:

- `grid-column` shorthand for `grid-column-start` and `grid-column-end`
- `grid-row` shorthand for `grid-row-start` and `grid-row-end`

To see this in action, download the line-based placement starting point file or see it live here. It has a defined grid and a simple article outlined. You can see that *auto-placement* is placing each item into its own cell in the grid.

Let's arrange all of the elements for our site by using the grid lines. Add the following rules to the bottom of your CSS:

CSS Play

```
header {
  grid-column: 1 / 3;
  grid-row: 1;
}

article {
  grid-column: 2;
  grid-row: 2;
}

aside {
  grid-column: 1;
  grid-row: 2;
}

footer {
  grid-column: 1 / 3;
  grid-row: 3;
}
```

Play

> **Note:** You can also use the value `-1` to target the end column or row line, then count inwards from the end using negative values. Note also that lines count always from the edges of the explicit grid, not the implicit grid.

Positioning with grid-template-areas

An alternative way to arrange items on your grid is to use the `grid-template-areas` property and give the various elements of your design a name.

Remove the line-based positioning from the last example (or re-download the file to have a fresh starting point) and add the following CSS.

CSS Play

```
.container {
  display: grid;
  grid-template-areas:
    "header header"
    "sidebar content"
    "footer footer";
  grid-template-columns: 1fr 3fr;
  gap: 20px;
}

header {
  grid-area: header;
}

article {
  grid-area: content;
}

aside {
  grid-area: sidebar;
}
```

```
footer {
  grid-area: footer;
}
```

Reload the page and you will see that your items have been placed just as before without us needing to use any line numbers!

Play

The rules for `grid-template-areas` are as follows:

- You need to have every cell of the grid filled.
- To span across two cells, repeat the name.
- To leave a cell empty, use a `.` (period).
- Areas must be rectangular — for example, you can't have an L-shaped area.
- Areas can't be repeated in different locations.

You can play around with our layout, changing the footer to only sit underneath the article and the sidebar to span all the way down. This is a very nice way to describe a layout because it's clear just from looking at the CSS to know exactly what's happening.

Nesting grids and subgrid

It's possible to nest a grid within another grid, creating a "subgrid". You can do this by setting the `display: grid` property on a grid item.

Let's expand on the previous example by adding a container for articles and using a nested grid to control the layout of multiple articles. While we're using only one column in the nested grid, we can define the rows to be split in a 2:1:1 ratio by using the `grid-template-rows` property. This approach allows us to create a layout where one article at the top of the page has a large display, while the others have a smaller, preview-like layout.

CSS Play

```css
.articles {
  display: grid;
  grid-template-rows: 2fr 1fr 1fr;
  gap: inherit;
}

article {
  padding: 10px;
  border: 2px solid rgb(79 185 227);
  border-radius: 5px;
}
```

Play

To make it easier to work with layouts in nested grids, you can use `subgrid` on `grid-template-rows` and `grid-template-columns` properties. This allows you to leverage the tracks defined in the parent grid.

In the following example, we're using line-based placement, enabling the nested grid to span multiple columns and rows of the parent grid. We've added `subgrid` to inherit the parent grid's column tracks while adding a different layout for the rows within the nested grid.

HTML Play

```
<div class="container">
  <div>One</div>
  <div>Two</div>
  <div>Three</div>
  <div>Four</div>
  <div id="subgrid">
    <div>Five</div>
    <div>Six</div>
    <div>Seven</div>
    <div>Eight</div>
  </div>
  <div>Nine</div>
  <div>Ten</div>
</div>
```

CSS Play

```
.container {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  grid-template-rows: repeat(1, 1fr);
  gap: 10px;
}

#subgrid {
  grid-column: 1 / 4;
  grid-row: 2 / 4;
  display: grid;
  gap: inherit;
  grid-template-columns: subgrid;
  grid-template-rows: 2fr 1fr;
}
```

Play

Grid frameworks

Numerous grid frameworks are available, offering a 12 or 16-column grid, to help with laying out your content. The good news is that you probably won't need any third-party frameworks to help you create grid-based layouts — grid functionality is already included in the specification and is supported by most modern browsers.

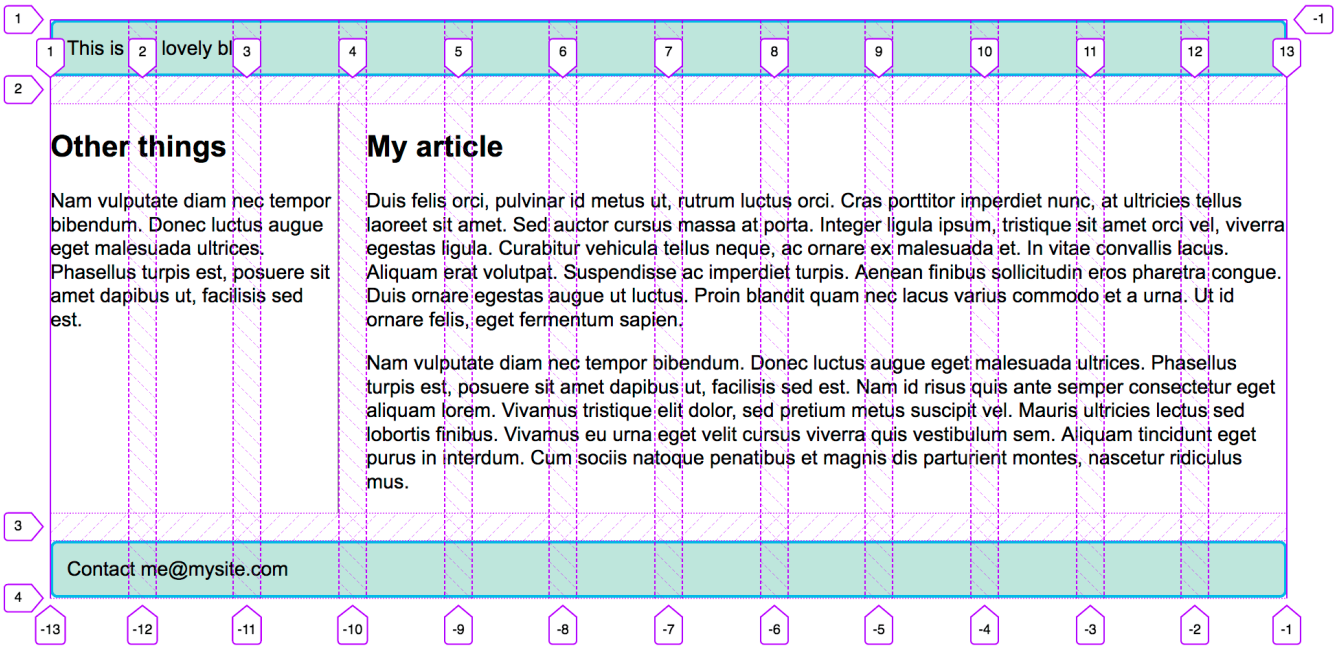
Download the starting point file . This has a container with a 12-column grid defined and the same markup we used in the previous two examples. We can now use line-based placement to place our content on the 12-column grid.

CSS Play

```
header {
  grid-column: 1 / 13;
  grid-row: 1;
}

article {
  grid-column: 4 / 13;
  grid-row: 2;
}

aside {
  grid-column: 1 / 4;
  grid-row: 2;
}

footer {
  grid-column: 1 / 13;
  grid-row: 3;
}
```

Play

If you use the Firefox Grid Inspector to overlay the grid lines on your design, you can see how our 12-column grid works.



Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Grid.

Summary

In this overview, we've toured the main features of CSS Grid Layout. You should be able to start using it in your designs. To dig further into the specification, read our guides on Grid Layout, which can be found below.

See also

- A list of guides related to the CSS grid layout
- Subgrid guide
- CSS grid inspector: Examine grid layouts on firefox-source-docs
- A complete guide to CSS grid , a visual guide on CSS-Tricks (2023)
- Grid Garden , an educational game to learn and better understand the basics of grid on cssgridgarden.com

Floats

Originally for floating images inside blocks of text, the `float` property became one of the most commonly used tools for creating multiple column layouts on webpages. With the advent of flexbox and grid it's now returned to its original purpose, as this article explains.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of How CSS works (study Introduction to CSS.)
Objective:	To learn how to create floated features on webpages and to use the clear property as well as other methods for clearing floats.

The background of floats

The `float` property was introduced to allow web developers to implement layouts involving an image floating inside a column of text, with the text wrapping around the left or right of it. The kind of thing you might get in a newspaper layout.

But web developers quickly realized that you can float anything, not just images, so the use of float broadened, for example, to fun layout effects such as drop-caps .

Floats have commonly been used to create entire website layouts featuring multiple columns of information floated so they sit alongside one another (the default behavior would be for the columns to sit below one another in the same order as they appear in the source). There are newer, better layout techniques available. Using floats in this way should be regarded as a legacy technique.

In this article we'll just concentrate on the proper uses of floats.

A float example

Let's explore the use of floats. We'll start with an example involving floating a block of text around an element. You can follow along by creating a new `index.html` file on your computer, filling it with an HTML template , and inserting the below code into it at the appropriate places. At the bottom of the section, you can see a live example of what the final code should look like.

First, we'll start off with some HTML. Add the following to your HTML body, removing anything that was inside there before:

HTML

```
<h1>Float example</h1>

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

<p>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus aliquam
  dolor, eu lacinia lorem placerat vulputate. Duis felis orci, pulvinar id metus
  ut, rutrum luctus orci. Cras porttitor imperdiet nunc, at ultricies tellus
  laoreet sit amet.
</p>

<p>
  Sed auctor cursus massa at porta. Integer ligula ipsum, tristique sit amet
  orci vel, viverra egestas ligula. Curabitur vehicula tellus neque, ac ornare
  ex malesuada et. In vitae convallis lacus. Aliquam erat volutpat. Suspendisse
  ac imperdiet turpis. Aenean finibus sollicitudin eros pharetra congue. Duis
  ornare egestas augue ut luctus. Proin blandit quam nec lacus varius commodo et
```

```
  a urna. Ut id ornare felis, eget fermentum sapien.
</p>

<p>
  Nam vulputate diam nec tempor bibendum. Donec luctus augue eget malesuada
  ultrices. Phasellus turpis est, posuere sit amet dapibus ut, facilisis sed
  est. Nam id risus quis ante semper consectetur eget aliquam lorem. Vivamus
  tristique elit dolor, sed pretium metus suscipit vel. Mauris ultricies lectus
  sed lobortis finibus. Vivamus eu urna eget velit cursus viverra quis
  vestibulum sem. Aliquam tincidunt eget purus in interdum. Cum sociis natoque
  penatibus et magnis dis parturient montes, nascetur ridiculus mus.
</p>
```

Now apply the following CSS to your HTML (using a `<style>` element or a `<link>` to a separate `.css` file — your choice):

CSS

```
body {
  width: 90%;
  max-width: 900px;
  margin: 0 auto;
  font:
    0.9em/1.2 Arial,
    Helvetica,
    sans-serif;
}

.box {
  width: 150px;
  height: 100px;
  border-radius: 5px;
  background-color: rgb(207 232 220);
  padding: 1em;
}
```

If you save and refresh, you'll see something much like what you'd expect: the box is sitting above the text, in normal flow.

Floating the box

To float the box, add the `float` and `margin-right` properties to the `.box` rule:

CSS Play

```
.box {
  float: left;
  margin-right: 15px;
  width: 150px;
  height: 100px;
  border-radius: 5px;
  background-color: rgb(207 232 220);
  padding: 1em;
}
```

Now if you save and refresh you'll see something like the following:

Play

Let's think about how the float works. The element with the float set on it (the `<div>` element in this case) is taken out of the normal layout flow of the document and stuck to the left-hand side of its parent container (`<body>` , in this case). Any content that would come below the floated element in the normal layout flow will now wrap around it instead, filling up the space to the right-hand side of it as far up as the top of the floated element. There, it will stop.

Floating the content to the right has exactly the same effect, but in reverse: the floated element will stick to the right, and the content will wrap around it to the left. Try changing the float value to `right` and replace `margin-right` with `margin-left` in the last ruleset to see what the result is.

Visualizing the float

While we can add a margin to the float to push the text away, we can't add a margin to the text to move it away from the float. This is because a floated element is taken out of normal flow and the boxes of the following items actually run behind the float. You can see this by making some changes to your example.

Add a class of `special` to the first paragraph of text, the one immediately following the floated box, then in your CSS add the following rules. These will give our following paragraph a background color.

CSS Play

```
.special {
  background-color: rgb(148 255 172);
  padding: 10px;
  color: purple;
}
```

To make the effect easier to see, change the `margin-right` on your float to `margin` so you get space all around the float. You'll be able to see the background on the paragraph running right underneath the floated box, as in the example below.

Play

The line boxes of our following element have been shortened so the text runs around the float, but due to the float being removed from normal flow the box around the paragraph still remains full width.

Clearing floats

We've seen that a float is removed from normal flow and that other elements will display beside it. If we want to stop the following element from moving up, we need to *clear* it; this is achieved with the `clear` property.

In your HTML from the previous example, add a class of `cleared` to the second paragraph below the floated item. Then add the following to your CSS:

CSS Play

```
.cleared {
  clear: left;
}
```

Play

You should see that the second paragraph now clears the floated element and no longer comes up alongside it. The `clear` property accepts the following values:

- `left` : Clear items floated to the left.
- `right` : Clear items floated to the right.
- `both` : Clear any floated items, left or right.

Clearing boxes wrapped around a float

You now know how to clear something following a floated element, but let's see what happens if you have a tall float and a short paragraph, with a box wrapped around *both* elements.

The problem

Change your document so that the first paragraph and the floated box are jointly wrapped with a `<div>` , which has a class of `wrapper` .

HTML Play

```
<div class="wrapper">
  <div class="box">Float1</div>

  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
    aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
    pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc, at
    ultricies tellus laoreet sit amet.
  </p>
</div>
```

In your CSS, add the following rule for the `.wrapper` class and then reload the page:

CSS Play

```css
.wrapper {
  background-color: rgb(148 255 172);
  padding: 10px;
  color: purple;
}
```

In addition, remove the original `.cleared` class:

CSS

```css
.cleared {
  clear: left;
}
```

You'll see that, just like in the example where we put a background color on the paragraph, the background color runs behind the float.

Play

Once again, this is because the float has been taken out of normal flow. You might expect that by wrapping the floated box and the text of first paragraph that wraps around the float together, the subsequent content will be cleared of the box. But this is not the case, as shown above. To deal with this, the standard method is to create a block formatting context (BFC) using the `display` property.

display: flow-root

To solve this problem is to use the value `flow-root` of the `display` property. This exists only to create a BFC without using hacks — there will be no unintended consequences when you use it.

CSS Play

```
.wrapper {
  background-color: rgb(148 255 172);
  padding: 10px;
  color: purple;
  display: flow-root;
}
```

Play

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Floats.

Summary

You now know all there is to know about floats in modern web development. See the article on legacy layout methods for information on how they used to be used, which may be useful if you find yourself working on older projects.

Positioning

Positioning allows you to take elements out of normal document flow and make them behave differently, for example, by sitting on top of one another or by always remaining in the same place inside the browser viewport. This article explains the different `position` values and how to use them.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of How CSS works (study Introduction to CSS.)
Objective:	To learn how CSS positioning works.

We'd like you to do the following exercises on your local computer. If possible, grab a copy of `0_basic-flow.html` from our GitHub repo (source code here) and use that as a starting point.

Introducing positioning

Positioning allows us to produce interesting results by overriding normal document flow. What if you want to slightly alter the position of some boxes from their default flow position to give a slightly quirky, distressed feel? Positioning is your tool. Or what if you want to create a UI element that floats over the top of other parts of the page and/or always sits in the same place inside the browser window no matter how much the page is scrolled? Positioning makes such layout work possible.

There are a number of different types of positioning that you can put into effect on HTML elements. To make a specific type of positioning active on an element, we use the `position` property.

Static positioning

Static positioning is the default that every element gets. It just means "put the element into its normal position in the document flow — nothing special to see here."

To see this (and get your example set up for future sections) first add a `class` of `positioned` to the second `<p>` in the HTML:

HTML

```
<p class="positioned">…</p>
```

Now add the following rule to the bottom of your CSS:

CSS

```
.positioned {
  position: static;
  background: yellow;
}
```

If you save and refresh, you'll see no difference at all, except for the updated background color of the 2nd paragraph. This is fine — as we said before, static positioning is the default behavior!

> **Note:** You can see the example at this point live at `1_static-positioning.html` (see source code).

Relative positioning

Relative positioning is the first position type we'll take a look at. This is very similar to static positioning, except that once the positioned element has taken its place in the normal flow, you can then modify its final position, including making it overlap other elements on the page. Go ahead and update the `position` declaration in your code:

CSS

```
position: relative;
```

If you save and refresh at this stage, you won't see a change in the result at all. So how do you modify the element's position? You need to use the `top`, `bottom`, `left`, and `right` properties, which we'll explain in the next section.

Introducing top, bottom, left, and right

`top`, `bottom`, `left`, and `right` are used alongside `position` to specify exactly where to move the positioned element to. To try this out, add the following declarations to the `.positioned` rule in your CSS:

CSS Play

```
top: 30px;
left: 30px;
```

> **Note:** The values of these properties can take any units you'd reasonably expect: pixels, mm, rems, %, etc.

If you now save and refresh, you'll get a result something like this:

Play

Cool, huh? Ok, so this probably wasn't what you were expecting. Why has it moved to the bottom and to the right if we specified *top* and *left*? This may seem counterintuitive. You need to think of it as if there's an invisible force that pushes the specified side of the positioned box, moving it in the opposite direction. So, for example, if you specify `top: 30px;`, it's as if a force will push the top of the box, causing it to move downwards by 30px.

> **Note:** You can see the example at this point live at `2 relative-positioning.html` (see source code).

Absolute positioning

Absolute positioning brings very different results.

Setting position: absolute

Let's try changing the position declaration in your code as follows:

CSS Play

```
position: absolute;
```

If you now save and refresh, you should see something like so:

Play

First of all, note that the gap where the positioned element should be in the document flow is no longer there — the first and third elements have closed together like it no longer exists! Well, in a way, this is true. An absolutely positioned element no longer exists in the normal document flow. Instead, it sits on its own layer separate from everything else. This is very useful: it means that we can create isolated UI features that don't interfere with the layout of other elements on the page. For example, popup information boxes, control menus, rollover panels, UI features that can be dragged and dropped anywhere on the page, and so on.

Second, notice that the position of the element has changed. This is because `top`, `bottom`, `left`, and `right` behave in a different way with absolute positioning. Rather than positioning the element based on its relative position within the normal document flow, they specify the distance the element should be from each of the containing element's sides. So in this case, we are saying that the absolutely positioned element should sit 30px from the top of the "containing element" and 30px from the left. (In this case, the "containing element" is the **initial containing block**. See the section below for more information)

> **Note:** You can use `top`, `bottom`, `left`, and `right` to resize elements if you need to. Try setting `top: 0; bottom: 0; left: 0; right: 0;` and `margin: 0;` on your positioned elements and see what happens! Put it back again afterwards...

> **Note:** Yes, margins still affect positioned elements. Margin collapsing doesn't, however.

> **Note:** You can see the example at this point live at `3 absolute-positioning.html` (see source code).

Positioning contexts

Which element is the "containing element" of an absolutely positioned element? This is very much dependent on the position property of the ancestors of the positioned element (See Identifying the containing block).

If no ancestor elements have their position property explicitly defined, then by default all ancestor elements will have a static position. The result of this is the absolutely positioned element will be contained in the **initial containing block**. The initial containing block has the dimensions of the viewport and is also the block that contains the `<html>` element. In other words, the absolutely positioned element will be displayed outside of the `<html>` element and be positioned relative to the initial viewport.

The positioned element is nested inside the `<body>` in the HTML source, but in the final layout it's 30px away from the top and the left edges of the page. We can change the **positioning context**, that is, which element the absolutely positioned element is positioned relative to. This is done by setting positioning on one of the element's ancestors: to one of the elements it's nested inside of (you can't position it relative to an element it's not nested inside of). To see this, add the following declaration to your `body` rule:

CSS Play

```
position: relative;
```

This should give the following result:

Play

The positioned element now sits relative to the `<body>` element.

> **Note:** You can see the example at this point live at `4_positioning-context.html` (see source code).

Introducing z-index

All this absolute positioning is good fun, but there's another feature we haven't considered yet. When elements start to overlap, what determines which elements appear over others and which elements appear under others? In the example we've seen so far,

we only have one positioned element in the positioning context, and it appears on the top since positioned elements win over non-positioned elements. What about when we have more than one?

Try adding the following to your CSS to make the first paragraph absolutely positioned too:

CSS Play

```
p:nth-of-type(1) {
  position: absolute;
  background: lime;
  top: 10px;
  right: 30px;
}
```

At this point you'll see the first paragraph colored lime, moved out of the document flow, and positioned a bit above from where it originally was. It's also stacked below the original `.positioned` paragraph where the two overlap. This is because the `.positioned` paragraph is the second paragraph in the source order, and positioned elements later in the source order win over positioned elements earlier in the source order.

Can you change the stacking order? Yes, you can, by using the `z-index` property. "z-index" is a reference to the z-axis. You may recall from previous points in the course where we discussed web pages using horizontal (x-axis) and vertical (y-axis) coordinates to work out positioning for things like background images and drop shadow offsets. For languages that run left to right, (0,0) is at the top left of the page (or element), and the x- and y-axes run across to the right and down the page.

Web pages also have a z-axis: an imaginary line that runs from the surface of your screen towards your face (or whatever else you like to have in front of the screen). `z-index` values affect where positioned elements sit on that axis; positive values move them higher up the stack, negative values move them lower down the stack. By default, positioned elements all have a `z-index` of `auto`, which is effectively 0.

To change the stacking order, try adding the following declaration to your `p:nth-of-type(1)` rule:

CSS Play

```
z-index: 1;
```

You should now see the lime paragraph on top:

Play

Note that `z-index` only accepts unitless index values; you can't specify that you want one element to be 23 pixels up the Z-axis — it doesn't work like that. Higher values will go above lower values and it's up to you what values you use. Using values of 2 or 3 would give the same effect as values of 300 or 40000.

> **Note:** You can see an example for this live at `5 z-index.html` (see source code).

Fixed positioning

Let's now look at fixed positioning. This works in exactly the same way as absolute positioning, with one key difference: whereas absolute positioning fixes an element in place relative to its nearest positioned ancestor (the initial containing block if there isn't one), **fixed positioning** *usually* fixes an element in place relative to the visible portion of the viewport. (An exception to this occurs if one of the element's ancestors is a fixed containing block because its transform property has a value other than *none*.) This means that you can create useful UI items that are fixed in place, like persistent navigation menus that are always visible no matter how much the page scrolls.

Let's put together a simple example to show what we mean. First of all, delete the existing `p:nth-of-type(1)` and `.positioned` rules from your CSS.

Now update the `body` rule to remove the `position: relative;` declaration and add a fixed height, like so:

CSS Play

```
body {
  width: 500px;
  height: 1400px;
  margin: 0 auto;
}
```

Now we're going to give the h1 element `position: fixed;` and have it sit at the top of the viewport. Add the following rule to your CSS:

CSS Play

```
h1 {
  position: fixed;
  top: 0;
  width: 500px;
  margin-top: 0;
  background: white;
  padding: 10px;
}
```

The `top: 0;` is required to make it stick to the top of the screen. We give the heading the same width as the content column and then a white background and some padding and margin so the content won't be visible underneath it.

If you save and refresh, you'll see a fun little effect of the heading staying fixed — the content appears to scroll up and disappear underneath it. But notice how some of the content is initially clipped under the heading. This is because the positioned heading no longer appears in the document flow, so the rest of the content moves up to the top. We could improve this by moving the paragraphs all down a bit. We can do this by setting some top margin on the first paragraph. Add this now:

CSS Play

```
p:nth-of-type(1) {
  margin-top: 60px;
}
```

You should now see the finished example:

Play

> **Note:** You can see an example for this live at `6 fixed-positioning.html` (see source code).

Sticky positioning

There is another position value available called `position: sticky`, which is somewhat newer than the others. This is basically a hybrid between relative and fixed position. It allows a positioned element to act like it's relatively positioned until it's scrolled to a certain threshold (e.g., 10px from the top of the viewport), after which it becomes fixed.

Basic example

Sticky positioning can be used, for example, to cause a navigation bar to scroll with the page until a certain point and then stick to the top of the page.

CSS Play

```
.positioned {
  position: sticky;
  top: 30px;
  left: 30px;
}
```

Play

Scrolling index

An interesting and common use of `position: sticky` is to create a scrolling index page where different headings stick to the top of the page as they reach it. The markup for such an example might look like so:

HTML Play

```
<h1>Sticky positioning</h1>

<dl>
  <dt>A</dt>
  <dd>Apple</dd>
  <dd>Ant</dd>
  <dd>Altimeter</dd>
  <dd>Airplane</dd>
  <dt>B</dt>
  <dd>Bird</dd>
  <dd>Buzzard</dd>
  <dd>Bee</dd>
  <dd>Banana</dd>
  <dd>Beanstalk</dd>
  <dt>C</dt>
  <dd>Calculator</dd>
  <dd>Cane</dd>
  <dd>Camera</dd>
  <dd>Camel</dd>
  <dt>D</dt>
  <dd>Duck</dd>
  <dd>Dime</dd>
  <dd>Dipstick</dd>
  <dd>Drone</dd>
  <dt>E</dt>
  <dd>Egg</dd>
  <dd>Elephant</dd>
  <dd>Egret</dd>
</dl>
```

The CSS might look as follows. In normal flow the `<dt>` elements will scroll with the content. When we add `position: sticky` to the `<dt>` element, along with a `top` value of 0, supporting browsers will stick the headings to the top of the viewport as they reach that position. Each subsequent header will then replace the previous one as it scrolls up to that position.

CSS Play

```
dt {
  background-color: black;
  color: white;
  padding: 10px;
  position: sticky;
  top: 0;
  left: 0;
  margin: 1em 0;
}
```

Play

Sticky elements are "sticky" relative to the nearest ancestor with a "scrolling mechanism", which is determined by its ancestors' position property.

> **Note:** You can see this example live at `7_sticky-positioning.html` (see source code).

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Positioning.

Summary

I'm sure you had fun playing with basic positioning. While it's not an ideal method to use for entire layouts, there are many specific objectives it's suited for.

See also

- The `position` property reference.
- Practical positioning examples, for some more useful ideas.

Multiple-column layout

The multiple-column layout specification provides you with a method for laying content out in columns, as you might see in a newspaper. This article explains how to use this feature.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of How CSS works (study Introduction to CSS.)
Objective:	To learn how to create multiple-column layout on webpages, such as you might find in a newspaper.

A basic example

Let's explore how to use multiple-column layout — often referred to as *multicol*. You can follow along by downloading the multicol starting point file and adding the CSS into the appropriate places. At the bottom of the section you can see an example of what the final code should look like.

A three-column layout

Our starting point file contains some very simple HTML: a wrapper with a class of `container`, inside of which is a heading and some paragraphs.

The `<div>` with a class of container will become our multicol container. We enable multicol by using one of two properties: `column-count` or `column-width`. The `column-count` property takes a number as its value and creates that number of columns. If you add the following CSS to your stylesheet and reload the page, you'll get three columns:

CSS Play

```
.container {
  column-count: 3;
}
```

The columns that you create have flexible widths — the browser works out how much space to assign each column.

Play

Setting column-width

Change your CSS to use `column-width` as follows:

CSS Play

```
.container {
  column-width: 200px;
}
```

The browser will now give you as many columns as it can of the size that you specify; any remaining space is then shared between the existing columns. This means that you won't get exactly the width that you specify unless your container is exactly divisible by that width.

Play

Styling the columns

The columns created by multicol cannot be styled individually. There's no way to make one column bigger than other columns or to change the background or text color of a single column. You have two opportunities to change the way that columns display:

- Changing the size of the gap between columns using the `column-gap`.
- Adding a rule between columns with `column-rule`.

Using your example above, change the size of the gap by adding a `column-gap` property. You can play around with different values — the property accepts any length unit.

Now add a rule between the columns with `column-rule`. In a similar way to the `border` property that you encountered in previous lessons, `column-rule` is a shorthand for `column-rule-color`, `column-rule-style`, and `column-rule-width`, and accepts the same values as `border`.

CSS Play

```
.container {
  column-count: 3;
  column-gap: 20px;
  column-rule: 4px dotted rgb(79 185 227);
}
```

Try adding rules of different styles and colors.

Play

Something to take note of is that the rule doesn't take up any width of its own. It lies across the gap you created with `column-gap`. To make more space on either side of the rule, you'll need to increase the `column-gap` size.

Spanning columns

You can cause an element to span across all the columns. In this case, the content breaks where the spanning element's introduced and then continues below the element, creating a new set of columns. To cause an element to span all the columns, specify the value of `all` for the `column-span` property.

> **Note:** It isn't possible to cause an element to span just *some* columns. The property can only have the values of `none` (which is the default) or `all`.

Play

Columns and fragmentation

The content of a multi-column layout is fragmented. It essentially behaves the same way as content behaves in paged media, such as when you print a webpage. When you turn your content into a multicol container, it fragments into columns. In order for the content to do this, it must *break*.

Fragmented boxes

Sometimes, this breaking will happen in places that lead to a poor reading experience. In the example below, I have used multicol to lay out a series of boxes, each of which has a heading and some text inside. The heading becomes separated from the text if the columns fragment between the two.

HTML Play

```
<div class="container">
  <div class="card">
    <h2>I am the heading</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
      aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
      pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc,
      at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
      Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula.
    </p>
  </div>

  <div class="card">
    <h2>I am the heading</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
      aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
      pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc,
      at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
      Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula.
    </p>
  </div>
```

```html
  <div class="card">
    <h2>I am the heading</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
      aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
      pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc,
      at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
      Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula.
    </p>
  </div>
  <div class="card">
    <h2>I am the heading</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
      aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
      pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc,
      at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
      Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula.
    </p>
  </div>

  <div class="card">
    <h2>I am the heading</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
      aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
      pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc,
      at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
      Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula.
    </p>
  </div>

  <div class="card">
    <h2>I am the heading</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
      aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
      pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc,
      at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
      Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula.
    </p>
  </div>

  <div class="card">
    <h2>I am the heading</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
      aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
      pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc,
      at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta.
      Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula.
    </p>
  </div>
</div>
```

CSS Play

```css
.container {
  column-width: 250px;
  column-gap: 20px;
}

.card {
```

```
  background-color: rgb(207 232 220);
  border: 2px solid rgb(79 185 227);
  padding: 10px;
  margin: 0 0 1em 0;
}
```

Play

Setting break-inside

To control this behavior, we can use properties from the CSS Fragmentation specification. This specification gives us properties to control the breaking of content in multicol and in paged media. For example, by adding the property `break-inside` with a value of `avoid` to the rules for `.card`. This is the container of the heading and text, so we don't want it fragmented.

CSS Play

```
.card {
  break-inside: avoid;
```

```
  background-color: rgb(207 232 220);
  border: 2px solid rgb(79 185 227);
  padding: 10px;
  margin: 0 0 1em 0;
}
```

The addition of this property causes the boxes to stay in one piece—they now do not *fragment* across the columns.

Play

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Multicol.

Summary

You now know how to use the basic features of multiple-column layout, another tool at your disposal when choosing a layout method for the designs you're building.

See also

- CSS Fragmentation
- Using multi-column layouts

Responsive design

Responsive web design (RWD) is a web design approach to make web pages render well on all screen sizes and resolutions while ensuring good usability. It is the way to design for a multi-device web. In this article, we'll help you understand some techniques that can be used to master it.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps and CSS building blocks.)
Objective:	To understand the fundamental purposed and CSS features used to implement responsive designs.

Precursor to responsive design: mobile web design

Before responsive web design became the standard approach for making websites work across different device types, web developers used to talk about mobile web design, mobile web development, or sometimes, mobile-friendly design. These are basically the same as responsive web design — the goals are to make sure that websites work well across devices with different physical attributes (screen size, resolution) in terms of layout, content (text and media), and performance.

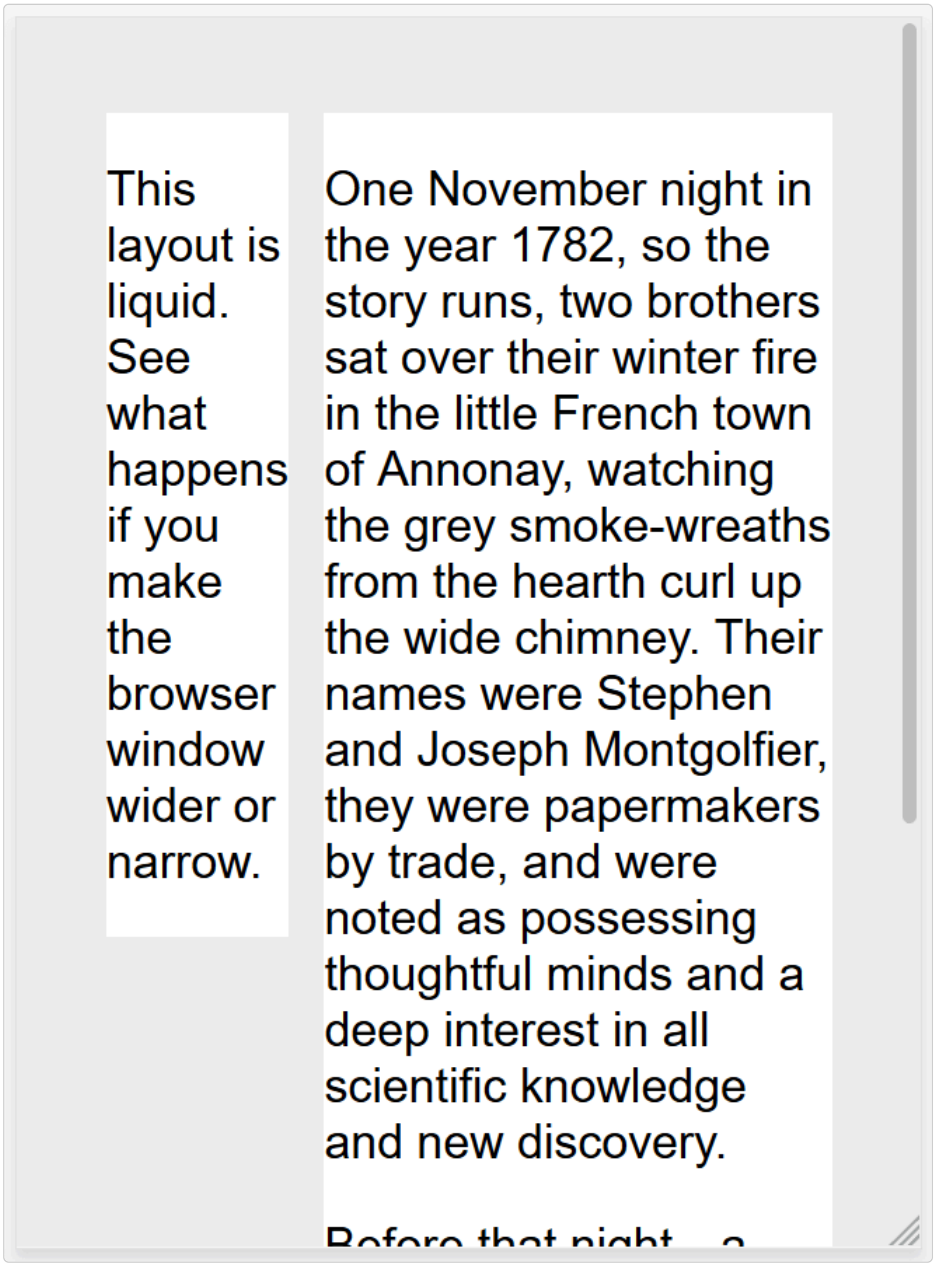
The difference is mainly to do with the devices involved, and the technologies available to create solutions:

- We used to talk about desktop or mobile, but now there are many different types of device available such as desktop, laptop, mobile, tablets, watches, etc. Instead of catering for a few different screen sizes, we now need to design sites defensively to cater for common screen sizes and resolutions, plus unknowns.
- Mobile devices used to be low-powered in terms of CPU/GPU and available bandwidth. Some didn't support CSS or even HTML, and as a result, it was common to perform server-side browser sniffing to determine device/browser type before then serving a site that the device would be able to cope with. Mobile devices often had really simple, basic experiences served to them because it was all they could handle. These days, mobile devices are able to handle the same technologies as desktop computers, so such techniques are less common.
  - You should still use the techniques discussed in this article to serve mobile users a suitable experience, as there are still constraints such as battery life and bandwidth to worry about.
  - User experience is also a concern. A mobile user of a travel site might just want to check flight times and delay information, for example, and not be presented with a 3D animated globe showing flight paths and your company history. This can be handled using responsive design techniques, however.
- Modern technologies are much better for creating responsive experiences. For example, responsive images/media technologies now allow appropriate media to be served to different devices without having to rely on techniques like server-side sniffing.

Introducing responsive web design

HTML is fundamentally responsive, or *fluid*. If you create a web page containing only HTML, with no CSS, and resize the window, the browser will automatically reflow the text to fit the viewport.

While the default responsive behavior may sound like no solution is needed, long lines of text displayed full screen on a wide monitor can be difficult to read. If wide screen line length is reduced with CSS, such as by creating columns or adding significant padding, the site may look squashed for the user who narrows their browser window or opens the site on a mobile device.



Creating a non-resizable web page by setting a fixed width doesn't work either; that leads to scroll bars on narrow devices and too much empty space on wide screens.

Responsive web design, or RWD, is a design approach that addresses the range of devices and device sizes, enabling automatic adaption to the screen, whether the content is viewed on a tablet, phone, television, or watch.

Responsive web design isn't a separate technology — it is an approach. It is a term used to describe a set of best practices used to create a layout that can *respond* to any device being used to view the content.

The term *responsive design*, coined by Ethan Marcotte in 2010 , described using fluid grids, fluid images, and media queries to create responsive content, as discussed in Zoe Mickley Gillenwater's book Flexible Web Design .

At the time, the recommendation was to use CSS `float` for layout and media queries to query the browser width, creating layouts for different breakpoints. Fluid images are set to not exceed the width of their container; they have their `max-width` property set to `100%`. Fluid images scale down when their containing column narrows but do not grow larger than their intrinsic size when the column grows. This enables an image to scale down to fit its content, rather than overflow it, but not grow larger and become pixelated if the container becomes wider than the image.

Modern CSS layout methods are inherently responsive, and, since the publication of Gillenwater's book and Marcotte's article, we have a multitude of features built into the web platform to make designing responsive sites easier.

The rest of this article will point you to the various web platform features you might want to use when creating a responsive site.

Media Queries

Media queries allow us to run a series of tests (e.g. whether the user's screen is greater than a certain width, or a certain resolution) and apply CSS selectively to style the page appropriately for the user's needs.

For example, the following media query tests to see if the current web page is being displayed as screen media (therefore not a printed document) and the viewport is at least `80rem` wide. The CSS for the `.container` selector will only be applied if these two things are true.

CSS

```
@media screen and (min-width: 80rem) {
  .container {
    margin: 1em 2em;
  }
}
```

You can add multiple media queries within a stylesheet, tweaking your whole layout or parts of it to best suit the various screen sizes. The points at which a media query is introduced, and the layout changed, are known as *breakpoints*.

A common approach when using Media Queries is to create a simple single-column layout for narrow-screen devices (e.g. mobile phones), then check for wider screens and implement a multiple-column layout when you know that you have enough screen width to handle it. Designing for mobile first is known as **mobile first** design.

If using breakpoints, best practices encourage defining media query breakpoints with relative units rather than absolute sizes of an individual device.

There are different approaches to the styles defined within a media query block; ranging from using media queries to `<link>` style sheets based on browser size ranges to only including custom properties variables to store values associated with each breakpoint.

Find out more in the MDN documentation for Media Queries.

Media queries can help with RWD, but are not a requirement. Flexible grids, relative units, and minimum and maximum unit values can be used without queries.

Responsive layout technologies

Responsive sites are built on flexible grids, meaning you don't need to target every possible device size with pixel perfect layouts.

By using a flexible grid, you can change a feature or add in a breakpoint and change the design at the point where the content starts to look bad. For example, to ensure line lengths don't become unreadably long as the screen size increases you can use `columns`; if a box becomes squashed with two words on each line as it narrows you can set a breakpoint.

Several layout methods, including Multiple-column layout, Flexbox, and Grid are responsive by default. They all assume that you are trying to create a flexible grid and give you easier ways to do so.

Multicol

With multicol, you specify a `column-count` to indicate the maximum number of columns you want your content to be split into. The browser then works out the size of these, a size that will change according to the screen size.

CSS

```
.container {
  column-count: 3;
}
```

If you instead specify a `column-width`, you are specifying a *minimum* width. The browser will create as many columns of that width as will comfortably fit into the container, then share out the remaining space between all the columns. Therefore the number of columns will change according to how much space there is.

CSS

```
.container {
  column-width: 10em;
}
```

You can use the `columns` shorthand to provide a maximum number of columns and a minimum column width. This can ensure line lengths don't become unreadably long as the screen size increases or too narrow as the screen size decreases.

Flexbox

In Flexbox, flex items shrink or grow, distributing space between the items according to the space in their container. By changing the values for `flex-grow` and `flex-shrink` you can indicate how you want the items to behave when they encounter more or less space around them.

In the example below the flex items will each take an equal amount of space in the flex container, using the shorthand of `flex: 1` as described in the layout topic Flexbox: Flexible sizing of flex items.

CSS

```
.container {
  display: flex;
}

.item {
  flex: 1;
}
```

> **Note:** As an example, we have built a simple responsive layout above using flexbox. We use a breakpoint to switch to multiple columns when the screen grows, and limit the size of the main content with `max-width`: example , source code .

CSS grid

In CSS Grid Layout the `fr` unit allows the distribution of available space across grid tracks. The next example creates a grid container with three tracks sized at `1fr`. This will create three column tracks, each taking one part of the available space in the container. You can find out more about this approach to create a grid in the Learn Layout Grids topic, under Flexible grids with the fr unit.

CSS

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

> **Note:** The grid layout version is even simpler as we can define the columns on the .wrapper: example , source code .

Responsive images/media

To ensure media is never larger than its responsive container, the following approach can be used:

CSS

```css
img,
picture,
video {
  max-width: 100%;
}
```

This scales media to ensure they never overflow their containers. Using a single large image and scaling it down to fit small devices wastes bandwidth by downloading images larger than what is needed.

Responsive Images, using the `<picture>` element and the `<img>` `srcset` and `sizes` attributes enables serving images targeted to the user's viewport and the device's resolution. For example, you can include a square image for mobile, but show the same scene as a landscape image on desktop.

The `<picture>` element enables providing multiple sizes along with "hints" (metadata that describes the screen size and resolution the image is best suited for), and the browser will choose the most appropriate image for each device, ensuring that a user will download an image size appropriate for the device they are using. Using `<picture>` along with `max-width` removes the need for sizing images with media queries. It enables targeting images with different aspect ratios to different viewport sizes.

You can also *art direct* images used at different sizes, thus providing a different crop or completely different image to different screen sizes.

You can find a detailed guide to Responsive Images in the Learn HTML section here on MDN.

Other useful tips:

- Always make sure to use an appropriate image format for your website images (such as PNG or JPG), and make sure to optimize the file size using a graphics editor before you put them on your website.
- You can make use of CSS features like gradients and shadows to implement visual effects without using images.
- You can use media queries inside the media attribute on `<source>` elements nested inside `<video>` / `<audio>` elements to serve video/audio files as appropriate for different devices (responsive video/audio).

Responsive typography

Responsive typography describes changing font sizes within media queries or using viewport units to reflect lesser or greater amounts of screen real estate.

Using media queries for responsive typography

In this example, we want to set our level 1 heading to be `4rem`, meaning it will be four times our base font size. That's a really large heading! We only want this jumbo heading on larger screen sizes, therefore we first create a smaller heading then use media queries to overwrite it with the larger size if we know that the user has a screen size of at least `1200px`.

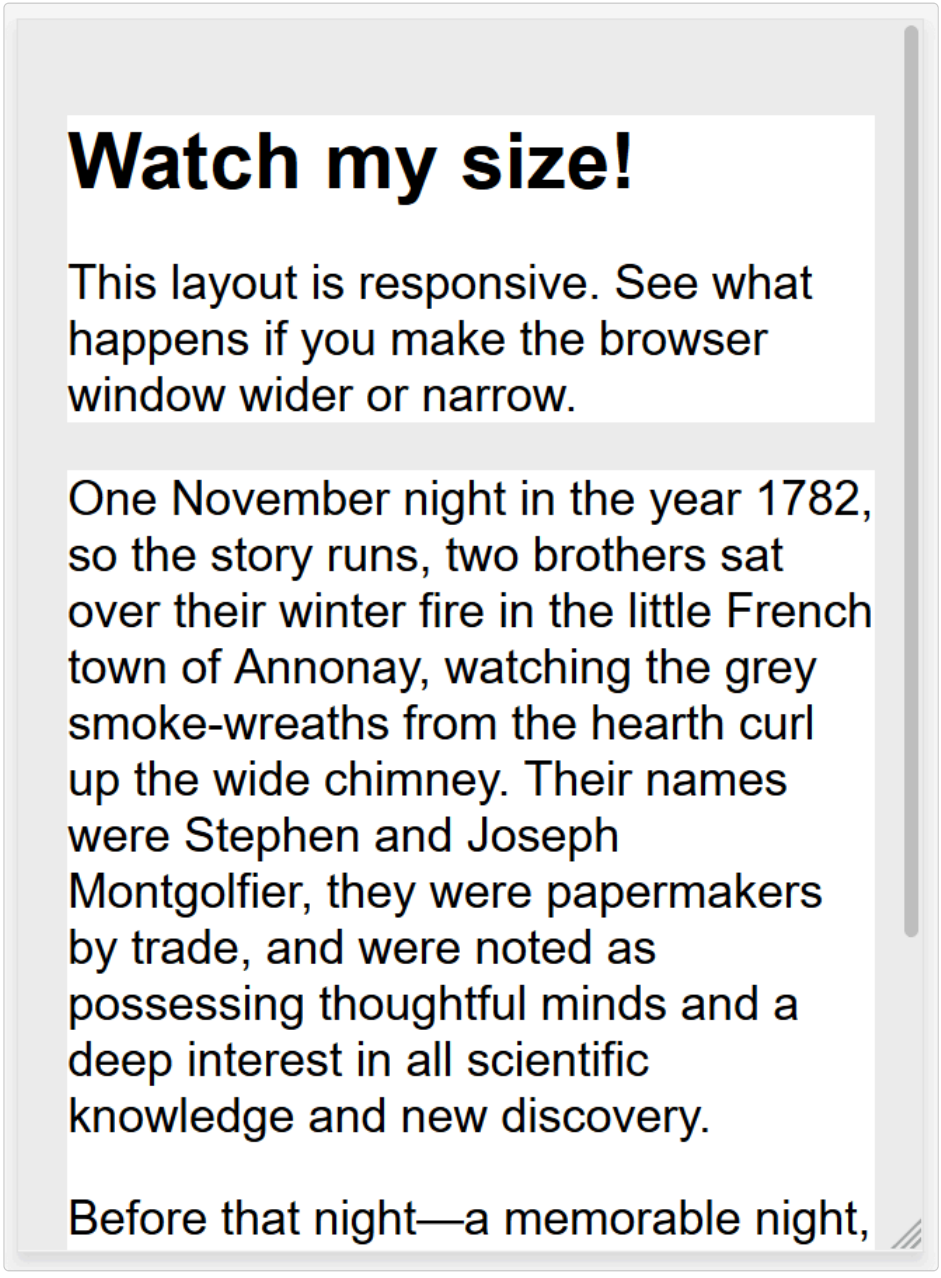
CSS

```css
html {
  font-size: 1em;
```

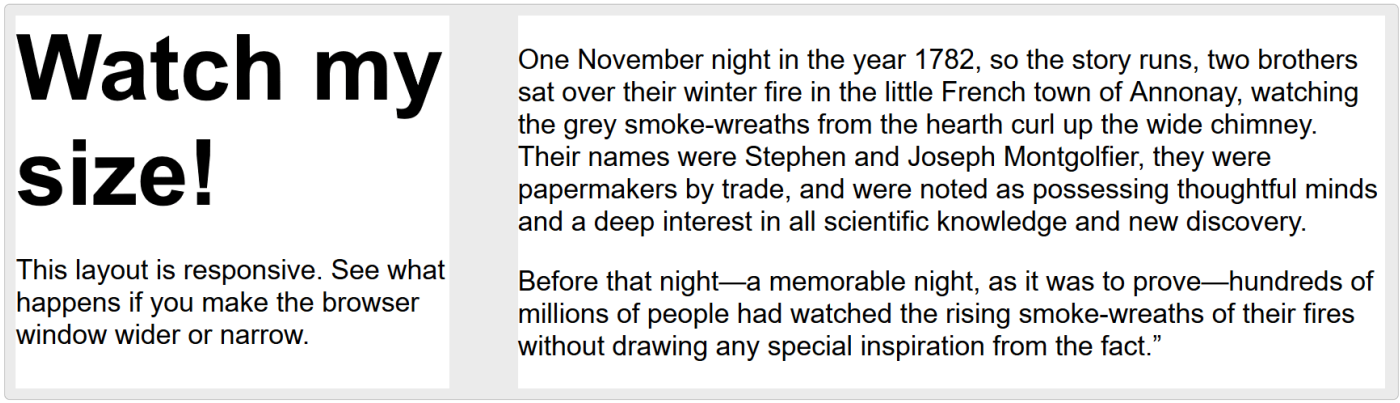
```
}

h1 {
  font-size: 2rem;
}

@media (min-width: 1200px) {
  h1 {
    font-size: 4rem;
  }
}
```

We have edited our responsive grid example above to also include responsive type using the method outlined. You can see how the heading switches sizes as the layout goes to the two column version.

On mobile the heading is smaller:



On desktop, however, we see the larger heading size:



> **Note:** See this example in action: example , source code .

As this approach to typography shows, you do not need to restrict media queries to only changing the layout of the page. They can be used to tweak any element to make it more usable or attractive at alternate screen sizes.

Using viewport units for responsive typography

Viewport units `vw` can also be used to enable responsive typography, without the need for setting breakpoints with media queries. `1vw` is equal to one percent of the viewport width, meaning that if you set your font size using `vw`, it will always relate to the size of the viewport.

CSS

```
h1 {
  font-size: 6vw;
}
```

The problem with doing the above is that the user loses the ability to zoom any text set using the `vw` unit, as that text is always related to the size of the viewport. **Therefore you should never set text using viewport units alone**.

There is a solution, and it involves using `calc()`. If you add the `vw` unit to a value set using a fixed size such as `em`s or `rem`s then the text will still be zoomable. Essentially, the `vw` unit adds on top of that zoomed value:

CSS

```
h1 {
  font-size: calc(1.5rem + 3vw);
}
```

This means that we only need to specify the font size for the heading once, rather than set it up for mobile and redefine it in the media queries. The font then gradually increases as you increase the size of the viewport.

> **Note:** See an example of this in action: example , source code .

The viewport meta tag

If you look at the HTML source of a responsive page, you will usually see the following `<meta>` tag in the `<head>` of the document.

HTML

```
<meta name="viewport" content="width=device-width,initial-scale=1" />
```

This viewport meta tag tells mobile browsers that they should set the width of the viewport to the device width, and scale the document to 100% of its intended size, which shows the document at the mobile-optimized size that you intended.

Why is this needed? Because mobile browsers tend to lie about their viewport width.

This meta tag exists because when smartphones first arrived, most sites were not mobile optimized. The mobile browser would, therefore, set the viewport width to 980 pixels, render the page at that width, and show the result as a zoomed-out version of the desktop layout. Users could zoom in and pan around the website to view the bits they were interested in, but it looked bad.

By setting `width=device-width` you are overriding a mobile device's default, like Apple's default `width=980px`, with the actual width of the device. Without it, your responsive design with breakpoints and media queries may not work as intended on mobile browsers. If you've got a narrow screen layout that kicks in at 480px viewport width or less, but the device is saying it is 980px wide, that user will not see your narrow screen layout.

So you should *always* include the viewport meta tag in the head of your documents.

Summary

Responsive design refers to a site or application design that responds to the environment in which it is viewed. It encompasses a number of CSS and HTML features and techniques and is now essentially just how we build websites by default. Consider the sites that you visit on your phone — it is probably fairly unusual to come across a site that is the desktop version scaled down, or where you need to scroll sideways to find things. This is because the web has moved to this approach of designing responsively.

It has also become much easier to achieve responsive designs with the help of the layout methods you have learned in these lessons. If you are new to web development today you have many more tools at your disposal than in the early days of responsive design. It is therefore worth checking the age of any materials you are using. While the historical articles are still useful, modern use of CSS and HTML makes it far easier to create elegant and useful designs, no matter what device your visitor views the site with.

See also

- Working with touchscreen devices:
  - Touch events provide the ability to interpret finger (or stylus) activity on touch screens or trackpads, enabling quality support for complex touch-based user interfaces.
  - Use the pointer or any-pointer media queries to load different CSS on touch-enabled devices.
- CSS-Tricks Guide to Media Queries

Beginner's guide to media queries

The **CSS Media Query** gives you a way to apply CSS only when the browser and device environment matches a rule that you specify, for example "viewport is wider than 480 pixels". Media queries are a key part of responsive web design, as they allow you to create different layouts depending on the size of the viewport, but they can also be used to detect other things about the environment your site is running on, for example whether the user is using a touchscreen rather than a mouse. In this lesson you will first learn about the syntax used in media queries, and then move on to use them in a working example showing how a simple design might be made responsive.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps and CSS building blocks.)
Objective:	To understand how to use media queries, and the most common approach for using them to create responsive designs.

Media Query Basics

The simplest media query syntax looks like this:

```css
@media media-type and (media-feature-rule) {
  /* CSS rules go here */
}
```

It consists of:

- A media type, which tells the browser what kind of media this code is for (e.g. print, or screen).
- A media expression, which is a rule, or test that must be passed for the contained CSS to be applied.
- A set of CSS rules that will be applied if the test passes and the media type is correct.

Media types

The possible types of media you can specify are:

- `all`
- `print`
- `screen`

The following media query will only set the body to 12pt if the page is printed. It will not apply when the page is loaded in a browser.

```css
@media print {
  body {
    font-size: 12pt;
  }
}
```

> **Note:** The media type here is different from the so-called MIME type.

> **Note:** There were a number of other media types defined in the Level 3 Media Queries specification; these have been deprecated and should be avoided.

> **Note:** Media types are optional; if you do not indicate a media type in your media query, then the media query will default to being for all media types.

Media feature rules

After specifying the type, you can then target a media feature with a rule.

Width and height

The feature we tend to detect most often in order to create responsive designs (and that has widespread browser support) is viewport width, and we can apply CSS if the viewport is above or below a certain width — or an exact width — using the `min-width`, `max-width`, and `width` media features.

These features are used to create layouts that respond to different screen sizes. For example, to change the body text color to red if the viewport is exactly 600 pixels, you would use the following media query.

CSS

```css
@media screen and (width: 600px) {
  body {
    color: red;
  }
}
```

Open this example in the browser, or view the source.

The `width` (and `height`) media features can be used as ranges, and therefore be prefixed with `min-` or `max-` to indicate that the given value is a minimum, or a maximum. For example, to make the color blue if the viewport is 600 pixels or narrower, use `max-width`:

CSS

```css
@media screen and (max-width: 600px) {
  body {
    color: blue;
  }
}
```

Open this example in the browser, or view the source.

In practice, using minimum or maximum values is much more useful for responsive design so you will rarely see `width` or `height` used alone.

There are many other media features that you can test for, although some of the newer features introduced in Levels 4 and 5 of the media queries specification have limited browser support. Each feature is documented on MDN along with browser support information, and you can find a complete list at Using Media Queries: Syntax.

Orientation

One well-supported media feature is `orientation`, which allows us to test for portrait or landscape mode. To change the body text color if the device is in landscape orientation, use the following media query.

CSS

```
@media (orientation: landscape) {
  body {
    color: rebeccapurple;
  }
}
```

Open this example in the browser, or view the source.

A standard desktop view has a landscape orientation, and a design that works well in this orientation may not work as well when viewed on a phone or tablet in portrait mode. Testing for orientation can help you to create a layout which is optimized for devices in portrait mode.

Use of pointing devices

As part of the Level 4 specification, the `hover` media feature was introduced. This feature means you can test if the user has the ability to hover over an element, which essentially means they are using some kind of pointing device; touchscreen and keyboard navigation does not hover.

CSS

```
@media (hover: hover) {
  body {
    color: rebeccapurple;
  }
}
```

Open this example in the browser, or view the source.

If we know the user cannot hover, we could display some interactive features by default. For users who can hover, we might choose to make them available when a link is hovered over.

Also in Level 4 is the `pointer` media feature. This takes three possible values, `none`, `fine` and `coarse`. A `fine` pointer is something like a mouse or trackpad. It enables the user to precisely target a small area. A `coarse` pointer is your finger on a touchscreen. The value `none` means the user has no pointing device; perhaps they are navigating with the keyboard only or with voice commands.

Using `pointer` can help you to design better interfaces that respond to the type of interaction a user is having with a screen. For example, you could create larger hit areas if you know that the user is interacting with the device as a touchscreen.

Using ranged syntax

One common case is to check if the viewport width is between two values:

CSS

```
@media (min-width: 30em) and (max-width: 50em) {
  /* … */
}
```

If you want to improve the readability of this, you can use "range" syntax:

CSS

```
@media (30em <= width <= 50em) {
  /* … */
}
```

So in this case, styles are applied when the viewport width is between `30em` and `50em`. For more information on using this style, see Using Media Queries: Syntax improvements in Level 4

More complex media queries

With all of the different possible media queries, you may want to combine them, or create lists of queries — any of which could be matched.

"and" logic in media queries

To combine media features you can use `and` in much the same way as we have used `and` above to combine a media type and feature. For example, we might want to test for a `min-width` and `orientation`. The body text will only be blue if the viewport is at least 600 pixels wide and the device is in landscape mode.

CSS

```
@media screen and (min-width: 600px) and (orientation: landscape) {
  body {
    color: blue;
  }
}
```

Open this example in the browser, or view the source .

"or" logic in media queries

If you have a set of queries, any of which could match, then you can comma separate these queries. In the below example the text will be blue if the viewport is at least 600 pixels wide OR the device is in landscape orientation. If either of these things are true the query matches.

CSS

```
@media screen and (min-width: 600px), screen and (orientation: landscape) {
  body {
    color: blue;
  }
}
```

Open this example in the browser, or view the source .

"not" logic in media queries

You can negate an entire media query by using the `not` operator. This reverses the meaning of the entire media query. Therefore in this next example the text will only be blue if the orientation is portrait.

CSS

```
@media not all and (orientation: landscape) {
  body {
    color: blue;
  }
}
```

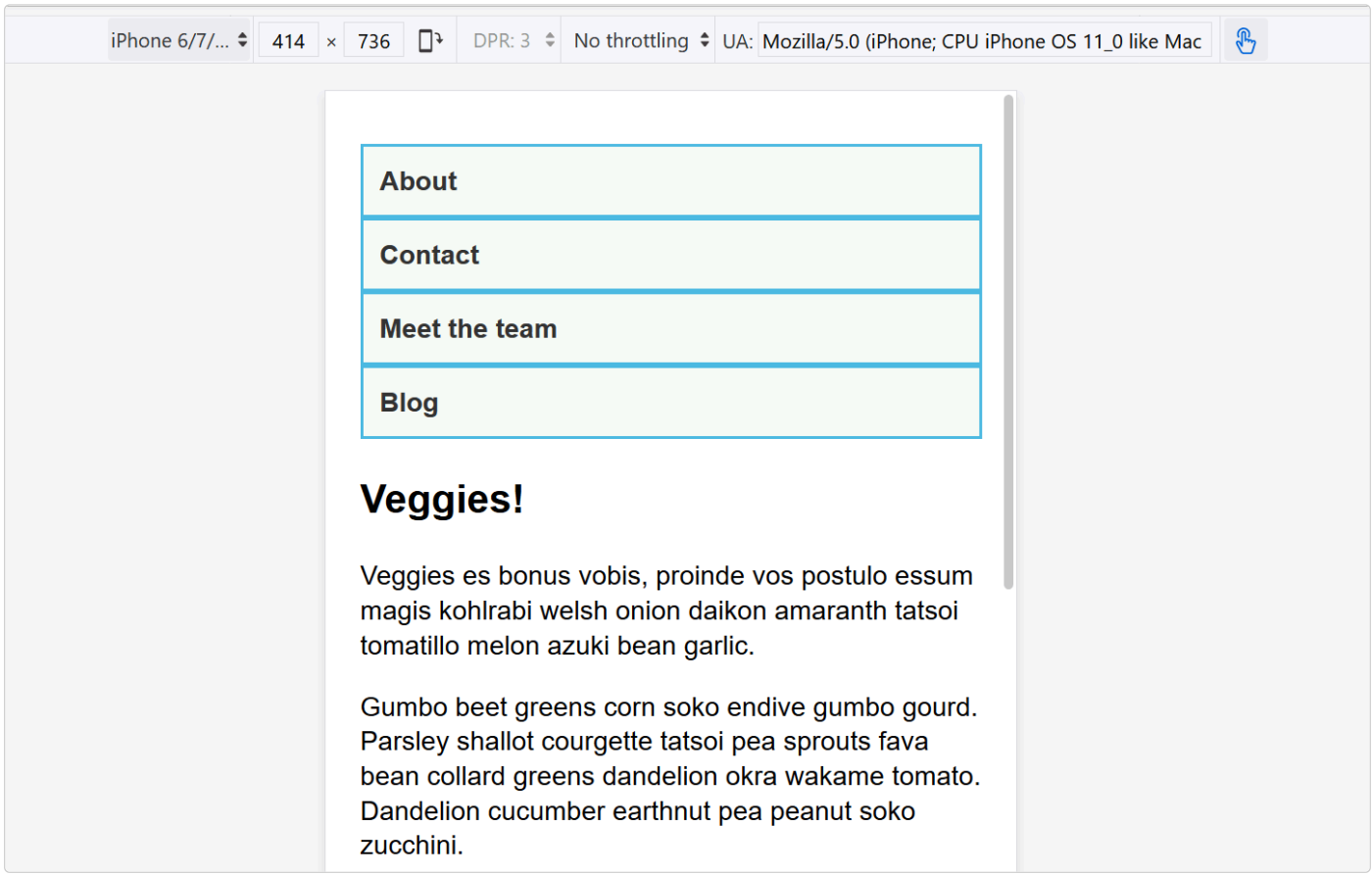
Open this example in the browser, or view the source .

How to choose breakpoints

In the early days of responsive design, many designers would attempt to target very specific screen sizes. Lists of the sizes of the screens of popular phones and tablets were published in order that designs could be created to neatly match those viewports.

There are now far too many devices, with a huge variety of sizes, to make that feasible. This means that instead of targeting specific sizes for all designs, a better approach is to change the design at the size where the content starts to break in some way. Perhaps the line lengths become far too long, or a boxed out sidebar gets squashed and hard to read. That's the point at which you want to use a media query to change the design to a better one for the space you have available. This approach means that it doesn't matter what the exact dimensions are of the device being used, every range is catered for. The points at which a media query is introduced are known as **breakpoints**.

The Responsive Design Mode in Firefox DevTools is very useful for working out where these breakpoints should go. You can easily make the viewport smaller and larger to see where the content would be improved by adding a media query and tweaking the design.



Active learning: mobile first responsive design

Broadly, you can take two approaches to a responsive design. You can start with your desktop or widest view and then add breakpoints to move things around as the viewport becomes smaller, or you can start with the smallest view and add layout as the viewport becomes larger. This second approach is described as **mobile first** responsive design and is quite often the best approach to follow.

The view for the very smallest devices is quite often a simple single column of content, much as it appears in normal flow. This means that you probably don't need to do a lot of layout for small devices — order your source well and you will have a readable layout by default.

The below walkthrough takes you through this approach with a very simple layout. In a production site you are likely to have more things to adjust within your media queries, however the approach would be exactly the same.

Walkthrough: a simple mobile-first layout

Our starting point is an HTML document with some CSS applied to add background colors to the various parts of the layout.

CSS

```css
* {
  box-sizing: border-box;
}

body {
  width: 90%;
  margin: 2em auto;
  font:
    1em/1.3 Arial,
    Helvetica,
    sans-serif;
}

a:link,
a:visited {
  color: #333;
}

nav ul,
aside ul {
  list-style: none;
  padding: 0;
}

nav a:link,
nav a:visited {
  background-color: rgb(207 232 220 / 20%);
  border: 2px solid rgb(79 185 227);
  text-decoration: none;
  display: block;
  padding: 10px;
  color: #333;
  font-weight: bold;
}

nav a:hover {
  background-color: rgb(207 232 220 / 70%);
}

.related {
  background-color: rgb(79 185 227 / 30%);
  border: 1px solid rgb(79 185 227);
  padding: 10px;
}

.sidebar {
  background-color: rgb(207 232 220 / 50%);
  padding: 10px;
}

article {
  margin-bottom: 1em;
}
```

We've made no layout changes, however the source of the document is ordered in a way that makes the content readable. This is an important first step and one which ensures that if the content were to be read out by a screen reader, it would be understandable.

HTML

```
<body>
  <div class="wrapper">
    <header>
      <nav>
        <ul>
          <li><a href="">About</a></li>
          <li><a href="">Contact</a></li>
          <li><a href="">Meet the team</a></li>
          <li><a href="">Blog</a></li>
        </ul>
      </nav>
    </header>
    <main>
      <article>
        <div class="content">
          <h1>Veggies!</h1>
          <p>…</p>
        </div>
        <aside class="related">
          <p>…</p>
        </aside>
      </article>

      <aside class="sidebar">
        <h2>External vegetable-based links</h2>
        <ul>
          <li>…</li>
        </ul>
      </aside>
    </main>

    <footer><p>&copy;2019</p></footer>
  </div>
</body>
```

This simple layout also works well on mobile. If we view the layout in Responsive Design Mode in DevTools we can see that it works pretty well as a straightforward mobile view of the site.

Open step 1 in the browser, or view the source .

If you want to follow on and implement this example as we go, make a local copy of step1.html on your computer.

From this point, start to drag the Responsive Design Mode view wider until you can see that the line lengths are becoming quite long, and we have space for the navigation to display in a horizontal line. This is where we'll add our first media query. We'll use ems, as this will mean that if the user has increased their text size, the breakpoint will happen at a similar line-length but wider viewport, than someone with a smaller text size.

Add the below code into the bottom of your step1.html CSS.

CSS

```
@media screen and (min-width: 40em) {
  article {
    display: grid;
    grid-template-columns: 3fr 1fr;
    column-gap: 20px;
  }

  nav ul {
```

```
    display: flex;
  }

  nav li {
    flex: 1;
  }
}
```

This CSS gives us a two-column layout inside the article, of the article content and related information in the aside element. We have also used flexbox to put the navigation into a row.

Open step 2 in the browser, or view the source .

Let's continue to expand the width until we feel there is enough room for the sidebar to also form a new column. Inside a media query we'll make the main element into a two column grid. We then need to remove the `margin-bottom` on the article in order that the two sidebars align with each other, and we'll add a `border` to the top of the footer. Typically these small tweaks are the kind of thing you will do to make the design look good at each breakpoint.

Again, add the below code into the bottom of your step1.html CSS.

CSS

```
@media screen and (min-width: 70em) {
  main {
    display: grid;
    grid-template-columns: 3fr 1fr;
    column-gap: 20px;
  }

  article {
    margin-bottom: 0;
  }

  footer {
    border-top: 1px solid #ccc;
    margin-top: 2em;
  }
}
```

Open step 3 in the browser, or view the source .

If you look at the final example at different widths you can see how the design responds and works as a single column, two columns, or three columns, depending on the available width. This is a very simple example of a mobile first responsive design.

The viewport meta tag

If you look at the HTML source in the above example, you'll see the following element included in the head of the document:

HTML

```
<meta name="viewport" content="width=device-width,initial-scale=1" />
```

This is the viewport meta tag — it exists as a way to control how mobile browsers render content. This is needed because by default, most mobile browsers lie about their viewport width. Non-responsive sites commonly look really bad when rendered in a narrow viewport, so mobile browsers usually render the site with a viewport width wider than the real device width by default (usually 980 pixels), and then shrink the rendered result so that it fits in the display.

This is all well and good, but it means that responsive sites are not going to work as expected. If the viewport width is reported as 980 pixels, then mobile layouts (for example created using a media query of `@media screen and (max-width: 600px) { }`) are not going to render as expected.

To remedy this, including a viewport meta tag like the one above on your page tells the browser "don't render the content with a 980 pixel viewport — render it using the real device width instead, and set a default initial scale level for better consistency." The media queries will then kick in as expected.

There are a number of other options you can put inside the `content` attribute of the viewport meta tag — see Using the viewport meta tag to control layout on mobile browsers for more details.

Do you really need a media query?

Flexbox, Grid, and multi-column layout all give you ways to create flexible and even responsive components without the need for a media query. It's always worth considering whether these layout methods can achieve what you want without adding media queries. For example, you might want a set of cards that are at least 200 pixels wide, with as many of these 200 pixels as will fit into the main article. This can be achieved with grid layout, using no media queries at all.

This could be achieved using the following:

HTML

```
<ul class="grid">
  <li>
    <h2>Card 1</h2>
    <p>…</p>
  </li>
  <li>
    <h2>Card 2</h2>
    <p>…</p>
  </li>
  <li>
    <h2>Card 3</h2>
    <p>…</p>
  </li>
  <li>
    <h2>Card 4</h2>
    <p>…</p>
  </li>
  <li>
    <h2>Card 5</h2>
    <p>…</p>
  </li>
</ul>
```

CSS

```
.grid {
  list-style: none;
  margin: 0;
  padding: 0;
  display: grid;
  gap: 20px;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
}

.grid li {
  border: 1px solid #666;
  padding: 10px;
}
```

Open the grid layout example in the browser, or view the source .

With the example open in your browser, make the screen wider and narrower to see the number of column tracks change. The nice thing about this method is that grid is not looking at the viewport width, but the width it has available for this component. It might seem strange to wrap up a section about media queries with a suggestion that you might not need one at all! However, in practice you will find that good use of modern layout methods, enhanced with media queries, will give the best results.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find a test to verify that you've retained this information before you move on — see Test your skills: Responsive web design and media queries.

Summary

In this lesson you have learned about media queries, and also discovered how to use them in practice to create a mobile first responsive design.

You could use the starting point that we have created to test out more media queries. For example, perhaps you could change the size of the navigation if you detect that the visitor has a coarse pointer, using the `pointer` media feature.

You could also experiment with adding different components and seeing whether the addition of a media query, or using a layout method like flexbox or grid is the most appropriate way to make the components responsive. Very often there is no right or wrong way — you should experiment and see which works best for your design and content.

Legacy layout methods

Grid systems are a very common feature used in CSS layouts, and before CSS Grid Layout they tended to be implemented using floats or other layout features. You imagine your layout as a set number of columns (e.g. 4, 6, or 12), and then fit your content columns inside these imaginary columns. In this article we'll explore how these older methods work, in order that you understand how they were used if you work on an older project.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of how CSS works (study Introduction to CSS and Styling boxes.)
Objective:	To understand the fundamental concepts behind the grid layout systems used prior to CSS Grid Layout being available in browsers.

Layout and grid systems before CSS Grid Layout

It may seem surprising to anyone coming from a design background that CSS didn't have an inbuilt grid system until very recently, and instead we seemed to be using a variety of sub-optimal methods to create grid-like designs. We now refer to these as "legacy" methods.

For new projects, in most cases CSS Grid Layout will be used in combination with one or more other modern layout methods to form the basis for any layout. You will however encounter "grid systems" using these legacy methods from time to time. It is worth understanding how they work, and why they are different to CSS Grid Layout.

This lesson will explain how grid systems and grid frameworks based on floats and flexbox work. Having studied Grid Layout you will probably be surprised how complicated this all seems! This knowledge will be helpful to you if you need to create fallback code for browsers that do not support newer methods, in addition to allowing you to work on existing projects which use these types of systems.

It is worth bearing in mind, as we explore these systems, that none of them actually create a grid in the way that CSS Grid Layout creates a grid. They work by giving items a size, and pushing them around to line them up in a way that *looks* like a grid.

A two column layout

Let's start with the simplest possible example — a two column layout. You can follow along by creating a new `index.html` file on your computer, filling it with a simple HTML template , and inserting the below code into it at the appropriate places. At the bottom of the section you can see a live example of what the final code should look like.

First of all, we need some content to put into our columns. Replace whatever is inside the body currently with the following:

HTML Play

```
<h1>2 column layout example</h1>
<div>
  <h2>First column</h2>
  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus
    aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci,
    pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc, at
    ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta. Integer
    ligula ipsum, tristique sit amet orci vel, viverra egestas ligula. Curabitur
```

```
    vehicula tellus neque, ac ornare ex malesuada et. In vitae convallis lacus.
    Aliquam erat volutpat. Suspendisse ac imperdiet turpis. Aenean finibus
    sollicitudin eros pharetra congue. Duis ornare egestas augue ut luctus.
    Proin blandit quam nec lacus varius commodo et a urna. Ut id ornare felis,
    eget fermentum sapien.
  </p>
</div>

<div>
  <h2>Second column</h2>
  <p>
    Nam vulputate diam nec tempor bibendum. Donec luctus augue eget malesuada
    ultrices. Phasellus turpis est, posuere sit amet dapibus ut, facilisis sed
    est. Nam id risus quis ante semper consectetur eget aliquam lorem. Vivamus
    tristique elit dolor, sed pretium metus suscipit vel. Mauris ultricies
    lectus sed lobortis finibus. Vivamus eu urna eget velit cursus viverra quis
    vestibulum sem. Aliquam tincidunt eget purus in interdum. Cum sociis natoque
    penatibus et magnis dis parturient montes, nascetur ridiculus mus.
  </p>
</div>
```

Each one of the columns needs an outer element to contain its content and let us manipulate all of it at once. In this example case we've chosen `<div>` s, but you could choose something more semantically appropriate like `<article>` s, `<section>` s, and `<aside>` , or whatever.

Now for the CSS. First, of all, apply the following to your HTML to provide some basic setup:

CSS Play

```
body {
  width: 90%;
  max-width: 900px;
  margin: 0 auto;
}
```

The body will be 90% of the viewport wide until it gets to 900px wide, in which case it will stay fixed at this width and center itself in the viewport. By default, its children (the h1 and the two `<div>` s) will span 100% of the width of the body. If we want the two `<div>` s to be floated alongside one another, we need to set their widths to total 100% of the width of their parent element or smaller so they can fit alongside one another. Add the following to the bottom of your CSS:

CSS Play

```
div:nth-of-type(1) {
  width: 48%;
}

div:nth-of-type(2) {
  width: 48%;
}
```

Here we've set both to be 48% of their parent's width — this totals 96%, leaving us 4% free to act as a gutter between the two columns, giving the content some space to breathe. Now we just need to float the columns, like so:

CSS Play

```
div:nth-of-type(1) {
  width: 48%;
  float: left;
}

div:nth-of-type(2) {
```

```
  width: 48%;
  float: right;
}
```

Putting this all together should give us a result like so:

Play

You'll notice here that we are using percentages for all the widths — this is quite a good strategy, as it creates a **liquid layout**, one that adjusts to different screen sizes and keeps the same proportions for the column widths at smaller screen sizes. Try adjusting the width of your browser window to see for yourself. This is a valuable tool for responsive web design.

> **Note:** You can see this example running at 0_two-column-layout.html (see also the source code).

Creating simple legacy grid frameworks

The majority of legacy frameworks use the behavior of the `float` property to float one column up next to another in order to create something that looks like a grid. Working through the process of creating a grid with floats shows you how this works and also introduces some more advanced concepts to build on the things you learned in the lesson on floats and clearing.

The easiest type of grid framework to create is a fixed width one — we just need to work out how much total width we want our design to be, how many columns we want, and how wide the gutters and columns should be. If we instead decided to lay out our design on a grid with columns that grow and shrink according to browser width, we would need to calculate percentage widths for the columns and gutters between them.

In the next sections we will look at how to create both. We will create a 12 column grid — a very common choice that is seen to be very adaptable to different situations given that 12 is nicely divisible by 6, 4, 3, and 2.

A simple fixed width grid

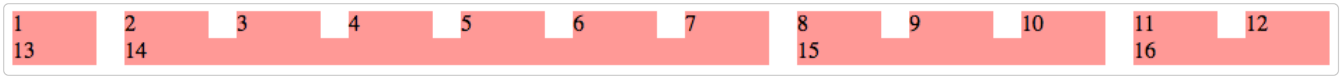
Lets first create a grid system that uses fixed width columns.

Start out by making a local copy of our sample simple-grid.html file, which contains the following markup in its body.

HTML

```
<div class="wrapper">
  <div class="row">
    <div class="col">1</div>
    <div class="col">2</div>
    <div class="col">3</div>
    <div class="col">4</div>
    <div class="col">5</div>
    <div class="col">6</div>
    <div class="col">7</div>
    <div class="col">8</div>
    <div class="col">9</div>
    <div class="col">10</div>
    <div class="col">11</div>
    <div class="col">12</div>
  </div>
  <div class="row">
    <div class="col span1">13</div>
    <div class="col span6">14</div>
    <div class="col span3">15</div>
    <div class="col span2">16</div>
  </div>
</div>
```

The aim is to turn this into a demonstration grid of two rows on a twelve column grid — the top row demonstrating the size of the individual columns, the second row some different sized areas on the grid.



In the <style> element, add the following code, which gives the wrapper container a width of 980 pixels, with padding on the right-hand side of 20 pixels. This leaves us with 960 pixels for our total column/gutter widths — in this case, the padding is subtracted from the total content width because we have set box-sizing to border-box on all elements on the site (see The alternative CSS box model for more explanation).

CSS

```
* {
  box-sizing: border-box;
}

body {
  width: 980px;
  margin: 0 auto;
}

.wrapper {
  padding-right: 20px;
}
```

Now use the row container that is wrapped around each row of the grid to clear one row from another. Add the following rule below your previous one:

CSS

```
.row {
  clear: both;
}
```

Applying this clearing means that we don't need to completely fill each row with elements making the full twelve columns. The rows will remain separated, and not interfere with each other.

The gutters between the columns are 20 pixels wide. We create these gutters as a margin on the left side of each column — including the first column, to balance out the 20 pixels of padding on the right-hand side of the container. So we have 12 gutters in total — 12 x 20 = 240.

We need to subtract that from our total width of 960 pixels, giving us 720 pixels for our columns. If we now divide that by 12, we know that each column should be 60 pixels wide.

Our next step is to create a rule for the class `.col`, floating it left, giving it a `margin-left` of 20 pixels to form the gutter, and a `width` of 60 pixels. Add the following rule to the bottom of your CSS:

CSS

```css
.col {
  float: left;
  margin-left: 20px;
  width: 60px;
  background: rgb(255 150 150);
}
```

The top row of single columns will now lay out neatly as a grid.

> **Note:** We've also given each column a light red color so you can see exactly how much space each one takes up.

Layout containers that we want to span more than one column need to be given special classes to adjust their `width` values to the required number of columns (plus gutters in between). We need to create an additional class to allow containers to span 2 to 12 columns. Each width is the result of adding up the column width of that number of columns plus the gutter widths, which will always number one less than the number of columns.

Add the following at the bottom of your CSS:

CSS

```css
/* Two column widths (120px) plus one gutter width (20px) */
.col.span2 {
  width: 140px;
}
/* Three column widths (180px) plus two gutter widths (40px) */
.col.span3 {
  width: 220px;
}
/* And so on… */
.col.span4 {
  width: 300px;
}
.col.span5 {
  width: 380px;
}
.col.span6 {
  width: 460px;
}
.col.span7 {
  width: 540px;
}
.col.span8 {
  width: 620px;
```

```
}
.col.span9 {
  width: 700px;
}
.col.span10 {
  width: 780px;
}
.col.span11 {
  width: 860px;
}
.col.span12 {
  width: 940px;
}
```

With these classes created we can now lay out different width columns on the grid. Try saving and loading the page in your browser to see the effects.

> **Note:** If you are having trouble getting the above example to work, try comparing it against our finished version on GitHub (see it running live also).

Try modifying the classes on your elements or even adding and removing some containers, to see how you can vary the layout. For example, you could make the second row look like this:

HTML

```
<div class="row">
  <div class="col span8">13</div>
  <div class="col span4">14</div>
</div>
```

Now you've got a grid system working, you can define the rows and the number of columns in each row, then fill each container with your required content. Great!

Creating a fluid grid

Our grid works nicely, but it has a fixed width. We really want a flexible (fluid) grid that will grow and shrink with the available space in the browser viewport. To achieve this we can turn the reference pixel widths into percentages.

The equation that turns a fixed width into a flexible percentage-based one is as follows.

```
target / context = result
```

For our column width, our **target width** is 60 pixels and our **context** is the 960 pixel wrapper. We can use the following to calculate a percentage.

```
60 / 960 = 0.0625
```

We then move the decimal point 2 places giving us a percentage of 6.25%. So, in our CSS we can replace the 60 pixel column width with 6.25%.

We need to do the same with our gutter width:

```
20 / 960 = 0.02083333333
```

So we need to replace the 20 pixel `margin-left` on our `.col` rule and the 20 pixel `padding-right` on `.wrapper` with 2.08333333%.

Updating our grid

To get started in this section, make a new copy of your previous example page, or make a local copy of our simple-grid-finished.html code to use as a starting point.

Update the second CSS rule (with the `.wrapper` selector) as follows:

```css
body {
  width: 90%;
  max-width: 980px;
  margin: 0 auto;
}

.wrapper {
  padding-right: 2.08333333%;
}
```

Not only have we given it a percentage `width`, we have also added a `max-width` property in order to stop the layout becoming too wide.

Next, update the fourth CSS rule (with the `.col` selector) like so:

```css
.col {
  float: left;
  margin-left: 2.08333333%;
  width: 6.25%;
  background: rgb(255 150 150);
}
```

Now comes the slightly more laborious part — we need to update all our `.col.span` rules to use percentages rather than pixel widths. This takes a bit of time with a calculator; to save you some effort, we've done it for you below.

Update the bottom block of CSS rules with the following:

```css
/* Two column widths (12.5%) plus one gutter width (2.08333333%) */
.col.span2 {
  width: 14.58333333%;
}
/* Three column widths (18.75%) plus two gutter widths (4.1666666) */
.col.span3 {
  width: 22.91666666%;
}
/* And so on… */
.col.span4 {
  width: 31.24999999%;
}
.col.span5 {
  width: 39.58333332%;
}
.col.span6 {
  width: 47.91666665%;
}
```

```
.col.span7 {
  width: 56.24999998%;
}
.col.span8 {
  width: 64.58333331%;
}
.col.span9 {
  width: 72.91666664%;
}
.col.span10 {
  width: 81.24999997%;
}
.col.span11 {
  width: 89.5833333%;
}
.col.span12 {
  width: 97.91666663%;
}
```

Now save your code, load it in a browser, and try changing the viewport width — you should see the column widths adjust nicely to suit.

> **Note:** If you are having trouble getting the above example to work, try comparing it against our finished version on GitHub (see it running live also).

Easier calculations using the calc() function

You could use the `calc()` function to do the math right inside your CSS — this allows you to insert simple mathematical equations into your CSS values, to calculate what a value should be. It is especially useful when there is complex math to be done, and you can even compute a calculation that uses different units, for example "I want this element's height to always be 100% of its parent's height, minus 50px". See this example from a MediaStream Recording API tutorial.

Anyway, back to our grids! Any column that spans more than one column of our grid has a total width of 6.25% multiplied by the number of columns spanned plus 2.08333333% multiplied by the number of gutters (which will always be the number of columns minus 1). The `calc()` function allows us to do this calculation right inside the width value, so for any item spanning 4 columns we can do this, for example:

CSS

```
.col.span4 {
  width: calc((6.25% * 4) + (2.08333333% * 3));
}
```

Try replacing your bottom block of rules with the following, then reload it in the browser to see if you get the same result:

CSS

```
.col.span2 {
  width: calc((6.25% * 2) + 2.08333333%);
}
.col.span3 {
  width: calc((6.25% * 3) + (2.08333333% * 2));
}
.col.span4 {
  width: calc((6.25% * 4) + (2.08333333% * 3));
}
.col.span5 {
  width: calc((6.25% * 5) + (2.08333333% * 4));
```

```
}
.col.span6 {
  width: calc((6.25% * 6) + (2.08333333% * 5));
}
.col.span7 {
  width: calc((6.25% * 7) + (2.08333333% * 6));
}
.col.span8 {
  width: calc((6.25% * 8) + (2.08333333% * 7));
}
.col.span9 {
  width: calc((6.25% * 9) + (2.08333333% * 8));
}
.col.span10 {
  width: calc((6.25% * 10) + (2.08333333% * 9));
}
.col.span11 {
  width: calc((6.25% * 11) + (2.08333333% * 10));
}
.col.span12 {
  width: calc((6.25% * 12) + (2.08333333% * 11));
}
```

> **Note:** You can see our finished version in fluid-grid-calc.html (also see it live).

Semantic versus "unsemantic" grid systems

Adding classes to your markup to define layout means that your content and markup becomes tied to your visual presentation. You will sometimes hear this use of CSS classes described as being "unsemantic" — describing how the content looks — rather than a semantic use of classes that describes the content. This is the case with our `span2`, `span3`, etc., classes.

These are not the only approach. You could instead decide on your grid and then add the sizing information to the rules for existing semantic classes. For example, if you had a `<div>` with a class of `content` on it that you wanted to span 8 columns, you could copy across the width from the `span8` class, giving you a rule like so:

CSS

```
.content {
  width: calc((6.25% * 8) + (2.08333333% * 7));
}
```

> **Note:** If you were to use a preprocessor such as Sass , you could create a simple mixin to insert that value for you.

Enabling offset containers in our grid

The grid we have created works well as long as we want to start all of the containers flush with the left-hand side of the grid. If we wanted to leave an empty column space before the first container — or between containers — we would need to create an offset class to add a left margin to our site to push it across the grid visually. More math!

Let's try this out.

Start with your previous code, or use our fluid-grid.html file as a starting point.

Let's create a class in our CSS that will offset a container element by one column width. Add the following to the bottom of your CSS:

CSS

```
.offset-by-one {
  margin-left: calc(6.25% + (2.08333333% * 2));
}
```

Or if you prefer to calculate the percentages yourself, use this one:

CSS

```
.offset-by-one {
  margin-left: 10.41666666%;
}
```

You can now add this class to any container you want to leave a one column wide empty space on the left-hand side of it. For example, if you have this in your HTML:

HTML

```
<div class="col span6">14</div>
```

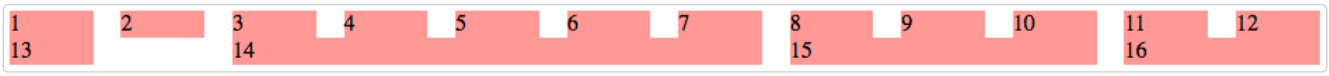
Try replacing it with

HTML

```
<div class="col span5 offset-by-one">14</div>
```

> **Note:** Notice that you need to reduce the number of columns spanned, to make room for the offset!

Try loading and refreshing to see the difference, or check out our fluid-grid-offset.html example (see it running live also). The finished example should look like this:



> **Note:** As an extra exercise, can you implement an `offset-by-two` class?

Floated grid limitations

When using a system like this you do need to take care that your total widths add up correctly, and that you don't include elements in a row that span more columns than the row can contain. Due to the way floats work, if the number of grid columns becomes too wide for the grid, the elements on the end will drop down to the next line, breaking the grid.

Also bear in mind that if the content of the elements gets wider than the rows they occupy, it will overflow and look a mess.

The biggest limitation of this system is that it is essentially one dimensional. We are dealing with columns, and spanning elements across columns, but not rows. It is very difficult with these older layout methods to control the height of elements without explicitly setting a height, and this is a very inflexible approach too — it only works if you can guarantee that your content will be a certain height.

Flexbox grids?

If you read our previous article about flexbox, you might think that flexbox is the ideal solution for creating a grid system. There are many flexbox-based grid systems available and flexbox can solve many of the issues that we've already discovered when creating our grid above.

However, flexbox was never designed as a grid system and poses a new set of challenges when used as one. As a simple example of this, we can take the same example markup we used above and use the following CSS to style the `wrapper`, `row`, and `col` classes:

CSS

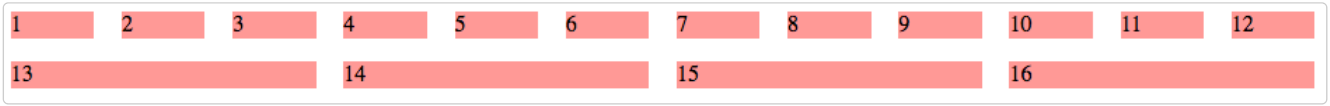
```
body {
  width: 90%;
  max-width: 980px;
  margin: 0 auto;
}

.wrapper {
  padding-right: 2.08333333%;
}

.row {
  display: flex;
}

.col {
  margin-left: 2.08333333%;
  margin-bottom: 1em;
  width: 6.25%;
  flex: 1 1 auto;
  background: rgb(255 150 150);
}
```

You can try making these replacements in your own example, or look at our flexbox-grid.html example code (see it running live also).

Here we are turning each row into a flex container. With a flexbox-based grid we still need rows in order to allow us to have elements that add up to less than 100%. We set that container to `display: flex`.

On `.col` we set the `flex` property's first value (`flex-grow`) to 1 so our items can grow, the second value (`flex-shrink`) to 1 so the items can shrink, and the third value (`flex-basis`) to `auto`. As our element has a `width` set, `auto` will use that width as the `flex-basis` value.

On the top line we get twelve neat boxes on the grid and they grow and shrink equally as we change the viewport width. On the next line, however, we only have four items and these also grow and shrink from that 60px basis. With only four of them they can grow a lot more than the items in the row above, the result being that they all occupy the same width on the second row.



To fix this we still need to include our `span` classes to provide a width that will replace the value used by `flex-basis` for that element.

They also don't respect the grid used by the items above because they don't know anything about it.

Flexbox is **one-dimensional** by design. It deals with a single dimension, that of a row or a column. We can't create a strict grid for columns and rows, meaning that if we are to use flexbox for our grid, we still need to calculate percentages as for the floated layout.

In your project you might still choose to use a flexbox 'grid' due to the additional alignment and space distribution capabilities flexbox provides over floats. You should, however, be aware that you are still using a tool for something other than what it was designed for. So you may feel like it is making you jump through additional hoops to get the end result you want.

Third party grid systems

Now that we understand the math behind our grid calculations, we are in a good place to look at some of the third party grid systems in common use. If you search for "CSS Grid framework" on the Web, you will find a huge list of options to choose from. Popular frameworks such as Bootstrap and Foundation include a grid system. There are also standalone grid systems, either developed using CSS or using preprocessors.

Let's take a look at one of these standalone systems as it demonstrates common techniques for working with a grid framework. The grid we will be using is part of Skeleton, a simple CSS framework.

To get started visit the Skeleton website , and choose "Download" to download the ZIP file. Unzip this and copy the skeleton.css and normalize.css files into a new directory.

Make a copy of our html-skeleton.html file and save it in the same directory as the skeleton and normalize CSS.

Include the skeleton and normalize CSS in the HTML page, by adding the following to its head:

HTML

```
<link href="normalize.css" rel="stylesheet" />
<link href="skeleton.css" rel="stylesheet" />
```

Skeleton includes more than a grid system — it also contains CSS for typography and other page elements that you can use as a starting point. We'll leave these at the defaults for now, however — it's the grid we are really interested in here.

> **Note:** Normalize is a really useful little CSS library written by Nicolas Gallagher, which automatically does some useful basic layout fixes and makes default element styling more consistent across browsers.

We will use similar HTML to our earlier example. Add the following into your HTML body:

HTML

```
<div class="container">
  <div class="row">
    <div class="col">1</div>
    <div class="col">2</div>
    <div class="col">3</div>
    <div class="col">4</div>
    <div class="col">5</div>
    <div class="col">6</div>
    <div class="col">7</div>
    <div class="col">8</div>
    <div class="col">9</div>
    <div class="col">10</div>
    <div class="col">11</div>
    <div class="col">12</div>
  </div>
  <div class="row">
    <div class="col">13</div>
    <div class="col">14</div>
    <div class="col">15</div>
    <div class="col">16</div>
```

```
  </div>
</div>
```

To start using Skeleton we need to give the wrapper `<div>` a class of `container` — this is already included in our HTML. This centers the content with a maximum width of 960 pixels. You can see how the boxes now never become wider than 960 pixels.

You can take a look in the skeleton.css file to see the CSS that is used when we apply this class. The `<div>` is centered using `auto` left and right margins, and a padding of 20 pixels is applied left and right. Skeleton also sets the `box-sizing` property to `border-box` like we did earlier, so the padding and borders of this element will be included in the total width.

CSS

```
.container {
  position: relative;
  width: 100%;
  max-width: 960px;
  margin: 0 auto;
  padding: 0 20px;
  box-sizing: border-box;
}
```

Elements can only be part of the grid if they are inside a row, so as with our earlier example we need an additional `<div>` or other element with a class of `row` nested between the content `<div>` elements and the container `<div>`. We've done this already as well.

Now let's lay out the container boxes. Skeleton is based on a 12 column grid. The top line boxes all need classes of `one column` to make them span one column.

Add these now, as shown in the following snippet:

HTML

```
<div class="container">
  <div class="row">
    <div class="one column">1</div>
    <div class="one column">2</div>
    <div class="one column">3</div>
    /* and so on */
  </div>
</div>
```

Next, give the containers on the second row classes explaining the number of columns they should span, like so:

HTML

```
<div class="row">
  <div class="one column">13</div>
  <div class="six columns">14</div>
  <div class="three columns">15</div>
  <div class="two columns">16</div>
</div>
```

Try saving your HTML file and loading it in your browser to see the effect.

> **Note:** If you are having trouble getting this example to work, try widening the window you're using to view it (the grid won't be displayed as described here if the window is too narrow). If that doesn't work, try comparing it to our html-skeleton-finished.html file (see it running live also).

If you look in the skeleton.css file you can see how this works. For example, Skeleton has the following defined to style elements with "three columns" classes added to them.

```css
.three.columns {
  width: 22%;
}
```

All Skeleton (or any other grid framework) is doing is setting up predefined classes that you can use by adding them to your markup. It's exactly the same as if you did the work of calculating these percentages yourself.

As you can see, we need to write very little CSS when using Skeleton. It deals with all of the floating for us when we add classes to our markup. It is this ability to hand responsibility for layout over to something else that made using a framework for a grid system a compelling choice! However these days, with CSS Grid Layout, many developers are moving away from these frameworks to use the inbuilt native grid that CSS provides.

Summary

You now understand how various grid systems are created, which will be useful in working with older sites and in understanding the difference between the native grid of CSS Grid Layout and these older systems.

Fundamental layout comprehension

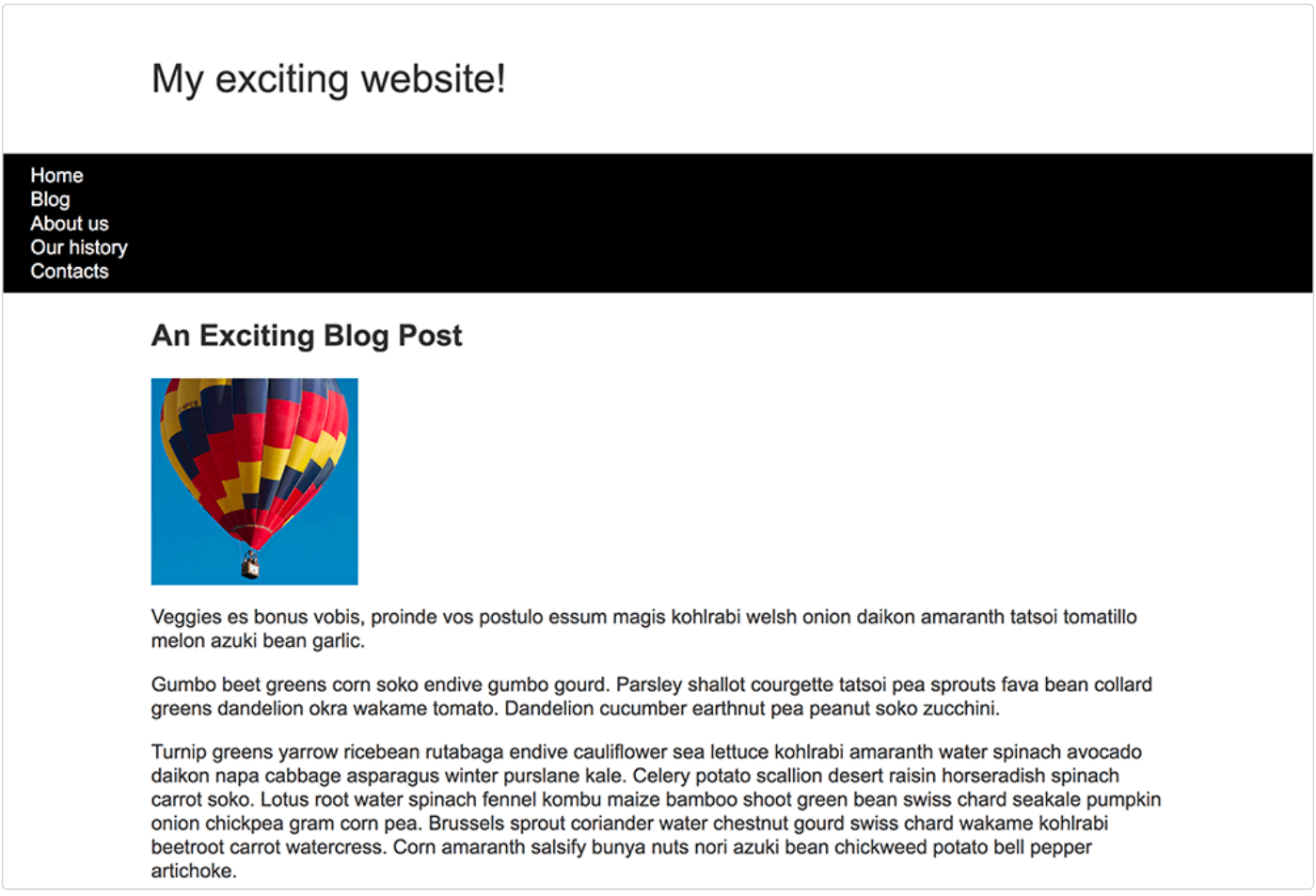
If you have worked through this module then you will have already covered the basics of what you need to know to do CSS layout today, and to work with older CSS as well. This task will test some of your knowledge by the way of developing a simple webpage layout using a variety of techniques.

Prerequisites:	Before attempting this assessment you should have already worked through all the articles in this module.
Objective:	To test comprehension of CSS layout methods using Flexbox, Grid, Floating and Positioning.

Starting point

You can download the HTML, CSS, and a set of six images here .

Save the HTML document and stylesheet into a directory on your computer, and add the images into a folder named `images`. Opening the `index.html` file in a browser should give you a page with basic styling but no layout, which should look something like the image below.



This starting point has all of the content of your layout as displayed by the browser in normal flow.

Alternatively, you could use an online editor such as CodePen , JSFiddle , or Glitch . If you use an online editor, you will need to upload the images and replace the values in the `src` attributes to point to the new image locations.

> **Note:** If you get stuck, you can reach out to us in one of our communication channels.

Project brief

You have been provided with some raw HTML, basic CSS, and images — now you need to create a layout for the design.

Your tasks

You now need to implement your layout. The tasks you need to achieve are:

1. To display the navigation items in a row, with an equal amount of space between the items.
2. The navigation bar should scroll with the content and then become stuck at the top of the viewport when it reaches it.
3. The image that is inside the article should have text wrapped around it.
4. The `<article>` and `<aside>` elements should display as a two column layout. The columns should be a flexible size so that if the browser window shrinks smaller the columns become narrower.
5. The photographs should display as a two column grid with a 1 pixel gap between the images.

Hints and tips

You will not need to edit the HTML in order to achieve this layout and the techniques you should use are:

- Flexbox
- Grid
- Floating
- Positioning

There are a few ways in which you could achieve some of these tasks, and there often isn't a single right or wrong way to do things. Try a few different approaches and see which works best. Make notes as you experiment.

Example

The following screenshot shows an example of what the finished layout for the design should look like:

My exciting website!

Home | Blog | About us | Our history | Contacts

An Exciting Blog Post

Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon azuki bean garlic.

Gumbo beet greens corn soko endive gumbo gourd. Parsley shallot courgette tatsoi pea sprouts fava bean collard greens dandelion okra wakame tomato. Dandelion cucumber earthnut pea peanut soko zucchini.

Turnip greens yarrow ricebean rutabaga endive cauliflower sea lettuce kohlrabi amaranth water spinach avocado daikon napa cabbage asparagus winter purslane kale. Celery potato scallion desert raisin horseradish spinach carrot soko. Lotus root water spinach fennel kombu maize bamboo shoot green bean swiss chard seakale pumpkin onion chickpea gram corn pea. Brussels sprout coriander water chestnut gourd swiss chard wakame kohlrabi beetroot carrot watercress. Corn amaranth salsify bunya nuts nori azuki bean chickweed potato bell pepper artichoke.

Nori grape silver beet broccoli kombu beet greens fava bean potato quandong celery. Bunya nuts black-eyed pea prairie turnip leek lentil turnip greens parsnip. Sea lettuce lettuce water chestnut eggplant winter purslane fennel azuki bean earthnut pea sierra leone bologi leek soko chicory celtuce parsley jícama salsify.

Celery quandong swiss chard chicory earthnut pea potato. Salsify taro catsear garlic gram celery bitterleaf wattle seed collard greens nori. Grape wattle seed kombu beetroot horseradish carrot squash brussels sprout chard.

Photography