HTML & CSS

HTML

Structuring the web with HTML

To build websites, you should know about HTML — the fundamental technology used to define the structure of a webpage. HTML is used to specify whether your web content should be recognized as a paragraph, list, heading, link, image, multimedia player, form, or one of many other available elements or even a new element that you define.

HTML (Hyper Text Markup Language) is a *markup language* that tells web browsers how to structure the web pages you visit. It can be as complicated or as simple as the web developer wants it to be. HTML consists of a series of elements, which you use to enclose, wrap, or *mark up* different parts of content to make it appear or act in a certain way. The enclosing tags can make content into a hyperlink to connect to another page, italicize words, and so on. For example, consider the following line of text:

```
My cat is very grumpy
```

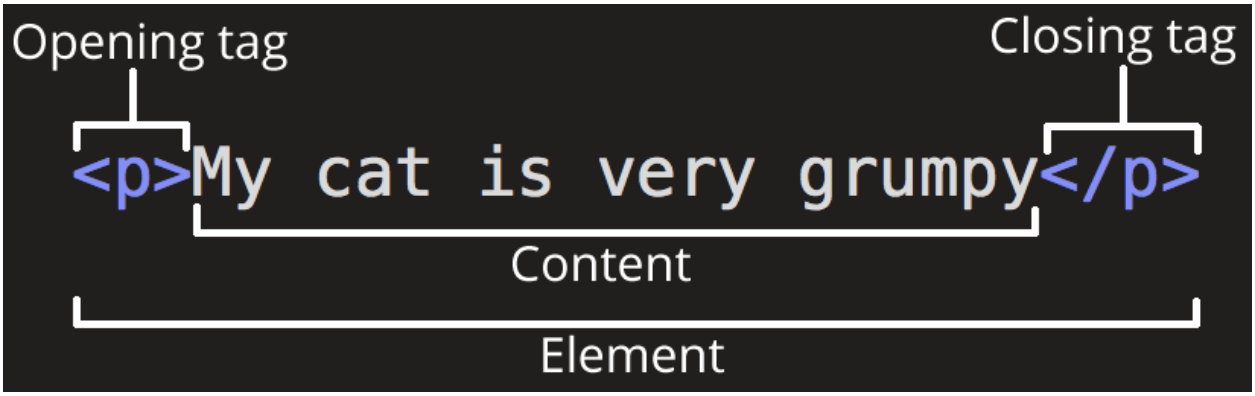
If we wanted the text to stand by itself, we could specify that it is a paragraph by enclosing it in a paragraph (`<p>`) element:

```
<p>My cat is very grumpy</p>
```

Note: Tags in HTML are not case-sensitive. This means they can be written in uppercase or lowercase. For example, a `<title>` tag could be written as `<title>`, `<TITLE>`, `<Title>`, `<TiTlE>`, etc., and it will work. However, it is best practice to write all tags in lowercase for consistency and readability.

Anatomy of an HTML element

Let's further explore our paragraph element from the previous section:



The anatomy of our element is:

- **The opening tag:** This consists of the name of the element (in this example, *p* for paragraph), wrapped in opening and closing angle brackets. This opening tag marks where the element begins or starts to take effect. In this example, it precedes the start of the paragraph text.
- **The content:** This is the content of the element. In this example, it is the paragraph text.
- **The closing tag:** This is the same as the opening tag, except that it includes a forward slash before the element name. This marks where the element ends. Failing to include a closing tag is a common beginner error that can produce peculiar results.

The element is the opening tag, followed by content, followed by the closing tag.

Nesting elements

Elements can be placed within other elements. This is called *nesting*. If we wanted to state that our cat is **very** grumpy, we could wrap the word *very* in a `<strong>` element, which means that the word is to have strong(er) text formatting:

```
<p>My cat is <strong>very</strong> grumpy.</p>
```

There is a right and wrong way to do nesting. In the example above, we opened the `p` element first, then opened the `strong` element. For proper nesting, we should close the `strong` element first, before closing the `p`.

Void elements

Not all elements follow the pattern of an opening tag, content, and a closing tag. Some elements consist of a single tag, which is typically used to insert/embed something in the document. Such elements are called void elements. For example, the `<img>` element embeds an image file onto a page:

```
<img
  src="https://raw.githubusercontent.com/mdn/beginner-html-site/gh-pages/images/firefox-icon.png"alt="Firefox icon" />
```

Note: In HTML, there is no requirement to add a `/` at the end of a void element's tag, for example: `<img src="images/cat.jpg" alt="cat" />`. However, it is also a valid syntax, and you may do this when you want your HTML to be valid XML.

Attributes

Elements can also have attributes. Attributes look like this:

Attributes contain extra information about the element that won't appear in the content. In this example, the `class` attribute is an identifying name used to target the element with style information.

An attribute should have:

- A space between it and the element name. (For an element with more than one attribute, the attributes should be separated by spaces too.)
- The attribute name, followed by an equal sign.
- An attribute value, wrapped with opening and closing quote marks.

Boolean attributes

Sometimes you will see attributes written without values. This is entirely acceptable. These are called Boolean attributes. Boolean attributes can only have one value, which is generally the same as the attribute name. For example, consider the `disabled` attribute, which you can assign to form input elements. (You use this to *disable* the form input elements so the user can't make entries. The disabled elements typically have a grayed-out appearance.) For example:

```
<input type="text" disabled="disabled" />
```

As shorthand, it is acceptable to write this as follows:

```
<!-- using the disabled attribute prevents the end user from
entering text into the input box -->
<input type="text" disabled /><!-- text input is allowed, as
it doesn't contain the disabled attribute -->
<input type="text" />
```

Omitting quotes around attribute values

If you look at code for a lot of other sites, you might come across a number of strange markup styles, including attribute values without quotes. This is permitted in certain circumstances, but it can also break your markup in other circumstances. The element in the code snippet below, `<a>`, is called an anchor. Anchors enclose text and turn them into links. The `href` attribute specifies the web address the link points to. You can write this basic version below with *only* the `href` attribute, like this:

```
<a href=https://www.mozilla.org/>favorite website</a>
```

Anchors can also have a `title` attribute, a description of the linked page. However, as soon as we add the `title` in the same fashion as the `href` attribute there are problems:

```
<a href=https://www.mozilla.org/ title=The Mozilla homepage>favorite website</a>
```

As written above, the browser misinterprets the markup, mistaking the `title` attribute for three attributes: a title attribute with the value `The`, and two Boolean attributes, `Mozilla` and `homepage`.

Single or double quotes?

In this article, you will also notice that the attributes are wrapped in double quotes. However, you might see single quotes in some HTML code. This is a matter of style. You can feel free to choose which one you prefer. Both of these lines are equivalent:

```
<a href='https://www.example.com'>A link to my example.</a><a href="https://www.example.com">A link to my example.</a>
```

Make sure you don't mix single quotes and double quotes.

Anatomy of an HTML document

```
<!doctype html><html lang="en-US"><head><meta charset="utf-8" /><title>My test page</title></head><body><p>This is my page</p></body></html>
```

Here we have:

1. `<!DOCTYPE html>`: The doctype. When HTML was young (1991-1992), doctypes were meant to act as links to a set of rules that the HTML page had to follow to be considered good HTML. Doctypes used to look something like this:More recently, the doctype is a historical artifact that needs to be included for everything else to

work right. `<!DOCTYPE html>` is the shortest string of characters that counts as a valid doctype. That is all you need to know!

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitiona
l//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitiona
l.dtd">
```

2. `<html></html>`: The `<html>` element. This element wraps all the content on the page. It is sometimes known as the root element.
3. `<head></head>`: The `<head>` element. This element acts as a container for everything you want to include on the HTML page, **that isn't the content** the page will show to viewers. This includes keywords and a page description that would appear in search results, CSS to style content, character set declarations, and more. You will learn more about this in the next article of the series.
4. `<meta charset="utf-8">`: The `<meta>` element. This element represents metadata that cannot be represented by other HTML meta-related elements, like `<base>`, `<link>`, `<script>`, `<style>` or `<title>`. The `charset` attribute specifies the character encoding for your document as UTF-8, which includes most characters from the vast majority of human written languages. With this setting, the page can now handle any textual content it might contain. There is no reason not to set this, and it can help avoid some problems later.
5. `<title></title>`: The `<title>` element. This sets the title of the page, which is the title that appears in the browser tab the page is loaded in. The page title is also used to describe the page when it is bookmarked.
6. `<body></body>`: The `<body>` element. This contains *all* the content that displays on the page, including text, images, videos, games, playable audio tracks, or whatever else.

Whitespace in HTML

In the examples above, you may have noticed that a lot of whitespace is included in the code. This is optional. These two code snippets are equivalent:

```
<p id="noWhitespace">Dogs are silly.</p>
<p id="whitespace">Dogs
```

```
    are
        silly.</p>
```

HTML comments

HTML has a mechanism to write comments in the code. Browsers ignore comments.

To write an HTML comment, wrap it in the special markers `<!--` and `-->`

What's in the head? Metadata in HTML

The head of an HTML document is the part that is not displayed in the web browser when the page is loaded. It contains information such as the page `<title>`, links to CSS (if you choose to style your HTML content with CSS), links to custom favicons, and other metadata (data about the HTML, such as the author, and important keywords that describe the document). Web browsers use information contained in the head to render the HTML document correctly. In this article we'll cover all of the above and more, in order to give you a good basis for working with markup.

What is the HTML head?

The HTML head is the contents of the `<head>` element. Unlike the contents of the `<body>` element (which are displayed on the page when loaded in a browser), the head's content is not displayed on the page. Instead, the head's job is to contain metadata about the document.

Adding a title

We've already seen the `<title>` element in action — this can be used to add a title to the document.

Metadata: the `<meta>` element

Metadata is data that describes data, and HTML has an "official" way of adding metadata to a document — the `<meta>` element. Of course, the other stuff we are talking about in this article could also be thought of as metadata too. There are a lot of different types of `<meta>` elements that can be included in your page's `<head>`, but we won't try to

explain them all at this stage, as it would just get too confusing. Instead, we'll explain a few things that you might commonly see, just to give you an idea.

Specifying your document's character encoding

In the example we saw above, this line was included:

```
<meta charset="utf-8" />
```

This element specifies the document's character encoding — the character set that the document is permitted to use. `utf-8` is a universal character set that includes pretty much any character from any human language. This means that your web page will be able to handle displaying any language; it's therefore a good idea to set this on every web page you create!

Note: Some browsers (like Chrome) automatically fix incorrect encodings, so depending on what browser you use, you may not see this problem. You should still set an encoding of `utf-8` on your page anyway to avoid any potential problems in other browsers.

Applying CSS and JavaScript to HTML

Just about all websites you'll use in the modern day will employ CSS to make them look cool, and JavaScript to power interactive functionality, such as video players, maps, games, and more. These are most commonly applied to a web page using the `<link>` element and the `<script>` element, respectively.

- The `<link>` element should always go inside the head of your document. This takes two attributes, `rel="stylesheet"`, which indicates that it is the document's stylesheet, and `href`, which contains the path to the stylesheet file:

  ```
  <link rel="stylesheet" href="my-css-file.css" />
  ```

- The `<script>` element should also go into the head, and should include a `src` attribute containing the path to the JavaScript you want to load, and `defer`, which basically instructs the browser to load the JavaScript after the page has finished parsing the HTML. This is useful as it makes sure that the HTML is all

loaded before the JavaScript runs, so that you don't get errors resulting from JavaScript trying to access an HTML element that doesn't exist on the page yet. There are actually a number of ways to handle loading JavaScript on your page, but this is the most reliable one to use for modern browsers (for others, read Script loading strategies).

```
<script src="my-js-file.js" defer></script>
```

Note: The `<script>` element may look like a void element, but it's not, and so needs a closing tag. Instead of pointing to an external script file, you can also choose to put your script inside the `<script>` element.

Active learning: applying CSS and JavaScript to a page

1. To start this active learning, grab a copy of our meta-example.html, script.js and style.css files, and save them on your local computer in the same directory. Make sure they are saved with the correct names and file extensions.
2. Open the HTML file in both your browser, and your text editor.
3. By following the information given above, add `<link>` and `<script>` elements to your HTML, so that your CSS and JavaScript are applied to your HTML.

If done correctly, when you save your HTML and refresh your browser you should be able to see that things have changed:

Setting the primary language of the document

Finally, it's worth mentioning that you can (and really should) set the language of your page. This can be done by adding the lang attribute to the opening HTML tag (as seen in the meta-example.html and shown below.)

```
<html lang="en-US">
  …
```

```
</html>
```

This is useful in many ways. Your HTML document will be indexed more effectively by search engines if its language is set (allowing it to appear correctly in language-specific results, for example), and it is useful to people with visual impairments using screen readers (for example, the word "six" exists in both French and English, but is pronounced differently.)

You can also set subsections of your document to be recognized as different languages. For example, we could set our Japanese language section to be recognized as Japanese, like so:

```
<p>Japanese example: <span lang="ja">ご飯が熱い。</span>.</p>
```

These codes are defined by the ISO 639-1 standard. You can find more about them in Language tags in HTML and XML.

HTML text fundamentals

The basics: headings and paragraphs

Most structured text consists of headings and paragraphs, whether you are reading a story, a newspaper, a college textbook, a magazine, etc.

Structured content makes the reading experience easier and more enjoyable.

In HTML, each paragraph has to be wrapped in a `<p>` element, like so:

```
<p>I am a paragraph, oh yes I am.</p>
```

Each heading has to be wrapped in a heading element:

```
<h1>I am the title of the story.</h1>
```

There are six heading elements: h1, h2, h3, h4, h5, and h6. Each element represents a different level of content in the document; `<h1>` represents the main heading, `<h2>` represents subheadings, `<h3>` represents sub-subheadings, and so on.

Implementing structural hierarchy

For example, in this story, the `<h1>` element represents the title of the story, the `<h2>` elements represent the title of each chapter, and the `<h3>` elements represent subsections of each chapter:

```
<h1>The Crushing Bore</h1><p>By Chris Mills</p><h2>Chapter 1: The dark night</h2><p>
  It was a dark night. Somewhere, an owl hooted. The rain lashed down on the…
</p><h2>Chapter 2: The eternal silence</h2><p>Our protagonist could not so much as a whisper out of the shadowy figure…</p>
<h3>The specter speaks</h3><p>
  Several more hours had passed, when all of a sudden the specter sat bolt
  upright and exclaimed, "Please have mercy on my soul!"
</p>
```

It's really up to you what the elements involved represent, as long as the hierarchy makes sense. You just need to bear in mind a few best practices as you create such structures:

- Preferably, you should use a single `<h1>` per page—this is the top level heading, and all others sit below this in the hierarchy.
- Make sure you use the headings in the correct order in the hierarchy. Don't use `<h3>` elements to represent subheadings, followed by `<h2>` elements to represent sub-subheadings—that doesn't make sense and will lead to weird results.

you could make any element *look* like a top level heading. Consider the following:

HTMLCopy to Clipboard

```
<span style="font-size: 32px; margin: 21px 0; display: bloc
k;">
  Is this a top level heading?
</span>
```

This is a `<span>` element. It has no semantics. You use it to wrap content when you want to apply CSS to it (or do something to it with JavaScript) without giving it any extra meaning.

Lists

Lists are everywhere on the web, too, and we've got three different types to worry about.

Unordered

Unordered lists are used to mark up lists of items for which the order of the items doesn't matter. Let's take a shopping list as an example:

```
milk
eggs
bread
hummus
```

Every unordered list starts off with a `<ul>` element—this wraps around all the list items:

```
<ul>
  milk
  eggs
  bread
  hummus
</ul>
```

The last step is to wrap each list item in a `<li>` (list item) element:

```
<ul><li>milk</li><li>eggs</li><li>bread</li><li>hummus</li></
ul>
```

Ordered

Ordered lists are lists in which the order of the items *does* matter. Let's take a set of directions as an example:

```
Drive to the end of the road
Turn right
Go straight across the first two roundabouts
Turn left at the third roundabout
The school is on your right, 300 meters up the road
```

The markup structure is the same as for unordered lists, except that you have to wrap the list items in an `<ol>` element, rather than `<ul>`:

HTMLCopy to Clipboard

```
<ol><li>Drive to the end of the road</li><li>Turn right</li><
li>Go straight across the first two roundabouts</li><li>Turn
left at the third roundabout</li><li>The school is on your ri
ght, 300 meters up the road</li></ol>
```

Description

Description lists enclose groups of terms and descriptions. Common uses for this element are implementing a glossary or displaying metadata (a list of key-value pairs). Let's consider some fishes with their interesting characteristics:

```
Albacore Tuna
The albacore is a very fast swimmer.
```

```
Asian Carp
Asian carp can consume 40% of their body weight in food a da
y!

Barracuda
Can grow to nearly 2 meters long!
```

The list is enclosed in a `<dl>` element, terms are enclosed in `<dt>` elements, and descriptions are enclosed in `<dd>` elements:

HTMLCopy to Clipboard

```
<dl><dt>Albacore Tuna</dt><dd>The albacore is a very fast swi
mmer.</dd><dt>Asian Carp</dt><dd>Asian carp can consume 40% o
f their body weight in food a day!</dd><dt>Barracuda</dt><dd>
Can grow to nearly 2 meters long!</dd></dl>
```

Description

Description lists enclose groups of terms and descriptions. Common uses for this element are implementing a glossary or displaying metadata (a list of key-value pairs). Let's consider some fishes with their interesting characteristics:

```
Albacore Tuna
The albacore is a very fast swimmer.

Asian Carp
Asian carp can consume 40% of their body weight in food a da
y!

Barracuda
Can grow to nearly 2 meters long!
```

The list is enclosed in a `<dl>` element, terms are enclosed in `<dt>` elements, and descriptions are enclosed in `<dd>` elements:

HTMLCopy to Clipboard

```
<dl><dt>Albacore Tuna</dt><dd>The albacore is a very fast swi
mmer.</dd><dt>Asian Carp</dt><dd>Asian carp can consume 40% o
f their body weight in food a day!</dd><dt>Barracuda</dt><dd>
Can grow to nearly 2 meters long!</dd></dl>
```

Italic, bold, underline…

The elements we've discussed so far have clear-cut associated semantics. The situation with `<b>`, `<i>`, and `<u>` is somewhat more complicated. They came about so people could write bold, italics, or underlined text in an era when CSS was still supported poorly or not at all. Elements like this, which only affect presentation and not semantics, are known as **presentational elements** and should no longer be used because, as we've seen before, semantics is so important to accessibility, SEO, etc.

HTML5 redefined `<b>`, `<i>`, and `<u>` with new, somewhat confusing, semantic roles.

Here's the best rule you can remember: It's only appropriate to use `<b>`, `<i>`, or `<u>` to convey a meaning traditionally conveyed with bold, italics, or underline when there isn't a more suitable element; and there usually is. Consider whether `<strong>`, `<em>`, `<mark>`, or `<span>` might be more appropriate.

Always keep an accessibility mindset. The concept of italics isn't very helpful to people using screen readers, or to people using a writing system other than the Latin alphabet.

- `<i>` is used to convey a meaning traditionally conveyed by italic: foreign words, taxonomic designation, technical terms, a thought…
- `<b>` is used to convey a meaning traditionally conveyed by bold: keywords, product names, lead sentence…
- `<u>` is used to convey a meaning traditionally conveyed by underline: proper name, misspelling…

Note: People strongly associate underlining with hyperlinks. Therefore, on the web, it's best to only underline links. Use the `<u>` element when it's semantically appropriate, but consider using CSS to change the default underline to something more appropriate on the web. The example below illustrates how it can be done.

HTMLPlayCopy to Clipboard

```
<!-- scientific names -->
<p>
  The Ruby-throated Hummingbird (<i>Archilochus colubris</i>)
is the most common
  hummingbird in Eastern North America.
</p><!-- foreign words -->
<p>
  The menu was a sea of exotic words like <i lang="uk-latn">v
atrushka</i>,
  <i lang="id">nasi goreng</i> and <i lang="fr">soupe à l'oig
non</i>.
</p><!-- a known misspelling -->
<p>Someday I'll learn how to <u class="spelling-error">spel</
u> better.</p><!-- term being defined when used in a definiti
on -->
<dl><dt>Semantic HTML</dt><dd>
    Use the elements based on their <b>semantic</b> meaning,
not their
    appearance.
  </dd></dl>
```

What is a hyperlink?

Hyperlinks are one of the most exciting innovations the Web has to offer. They've been a feature of the Web since the beginning, and are what makes the Web *a web.* Hyperlinks allow us to link documents to other documents or resources, link to specific parts of documents, or make apps available at a web address. Almost any web content can be converted to a link so that when clicked or otherwise activated the web browser goes to another web address (URL).

Note: A URL can point to HTML files, text files, images, text documents, video and audio files, or anything else that lives on the Web. If the web browser doesn't know how to display or handle the file, it will ask you if you want to open the file (in which case the duty of opening or handling the file is passed to a suitable native app on the device) or download the file (in which case you can try to deal with it later on).

Anatomy of a link

A basic link is created by wrapping the text or other content inside an `<a>` element and using the `href` attribute, also known as a **Hypertext Reference**, or **target**, that contains the web address.

HTMLCopy to Clipboard

```
<p>
  I'm creating a link to
  <a href="https://www.mozilla.org/en-US/">the Mozilla homepa
ge</a>.
</p>
```

This gives us the following result:

I'm creating a link to the Mozilla homepage.

HTMLPlayCopy to Clipboard

```
<a href="https://developer.mozilla.org/en-US/">
  <h1>MDN Web Docs</h1>
</a>
<p>
  Documenting web technologies, including CSS, HTML, and Java
Script, since 2005.
</p>
```

Image links

If you have an image you want to make into a link, use the `<a>` element to wrap the image file referenced with the `<img>` element. The example below uses a relative path to reference a locally stored SVG image file.

HTMLPlayCopy to Clipboard

```
<a href="https://developer.mozilla.org/en-US/"><img src="mdn_
logo.svg" alt="MDN Web Docs" /></a>
```

Absolute versus relative URLs

Two terms you'll come across on the Web are **absolute URL** and **relative URL:**

absolute URL: Points to a location defined by its absolute location on the web, including protocol and domain name. For example, if an `index.html` page is uploaded to a directory called `projects` that sits inside the **root** of a web server, and the website's domain is `https://www.example.com`, the page would be available at `https://www.example.com/projects/index.html` (or even just `https://www.example.com/projects/`, as most web servers just look for a landing page such as `index.html` to load if it isn't specified in the URL.)

An absolute URL will always point to the same location, no matter where it's used.

relative URL: Points to a location that is *relative* to the file you are linking from, more like what we looked at in the previous section. For example, if we wanted to link from our example file at `https://www.example.com/projects/index.html` to a PDF file in the same directory, the URL would just be the filename — `project-brief.pdf` — no extra information needed. If the PDF was available in a subdirectory inside `projects` called `pdfs`, the relative link would be `pdfs/project-brief.pdf` (the equivalent absolute URL would be `https://www.example.com/projects/pdfs/project-brief.pdf`.)

A relative URL will point to different places depending on the actual location of the file you refer from — for example if we moved our `index.html` file out of the `projects` directory and into the **root** of the website (the top level, not in any directories), the `pdfs/project-brief.pdf` relative URL link inside it would now point to a file located at `https://www.example.com/pdfs/project-brief.pdf`, not a file located at `https://www.example.com/projects/pdfs/project-brief.pdf`.

Of course, the location of the `project-brief.pdf` file and `pdfs` folder won't suddenly change because you moved the `index.html` file — this would make your link point to the wrong place, so it wouldn't work if clicked on. You need to be careful!

Active learning: creating a navigation menu

For this exercise, we'd like you to link some pages together with a navigation menu to create a multipage website. This is one common way in which a website is created — the same page structure is used on every page, including the same navigation menu, so

when links are clicked it gives the impression that you are staying in the same place, and different content is being brought up.

You'll need to make local copies of the following four pages, all in the same directory. For a complete file list, see the navigation-menu-start directory:

- index.html
- projects.html
- pictures.html
- social.html

Advanced text formatting

Description lists

In HTML text fundamentals, we walked through how to mark up basic lists in HTML, but we didn't mention the third type of list you'll occasionally come across — **description lists**. The purpose of these lists is to mark up a set of items and their associated descriptions, such as terms and definitions, or questions and answers. Let's look at an example of a set of terms and definitions

Description lists use a different wrapper than the other list types — `<dl>`; in addition each term is wrapped in a `<dt>` (description term) element, and each description is wrapped in a `<dd>` (description definition) element

Quotations

HTML also has features available for marking up quotations; which element you use depends on whether you are marking up a block or inline quotation.

Blockquotes

If a section of block level content (be it a paragraph, multiple paragraphs, a list, etc.) is quoted from somewhere else, you should wrap it inside a `<blockquote>` element to signify this, and include a URL pointing to the source of the quote inside a `cite` attribute. For example, the following markup is taken from the MDN `<blockquote>` element page:

HTMLPlayCopy to Clipboard

```
<p>
  The <strong>HTML <code>&lt;blockquote&gt;</code> Element</strong> (or
  <em>HTML Block Quotation Element</em>) indicates that the enclosed text is an
  extended quotation.
</p>
```

To turn this into a block quote, we would just do this:

HTMLPlayCopy to Clipboard

```
<p>Here is a blockquote:</p><blockquote
  cite="https://developer.mozilla.org/en-US/docs/Web/HTML/Element/blockquote"><p>
    The <strong>HTML <code>&lt;blockquote&gt;</code> Element</strong> (or
    <em>HTML Block Quotation Element</em>) indicates that the enclosed text is
    an extended quotation.
  </p></blockquote>
```

Inline quotations

Inline quotations work in exactly the same way, except that they use the `<q>` element. For example, the below bit of markup contains a quotation from the MDN `<q>` page:

HTMLPlayCopy to Clipboard

```
<p>
  The quote element — <code>&lt;q&gt;</code> — is
  <q cite="https://developer.mozilla.org/en-US/docs/Web/HTML/Element/q">
    intended for short quotations that don't require paragrap
```

```
h breaks.
  </q></p>
```

Browser default styling will render this as normal text put in quotes to indicate a quotation, like so:

Play

Citations

The content of the `cite` attribute sounds useful, but unfortunately browsers, screen readers, etc. don't really do much with it. There is no way to get the browser to display the contents of `cite`, without writing your own solution using JavaScript or CSS. If you want to make the source of the quotation available on the page you need to make it available in the text via a link or some other appropriate way.

There is a `<cite>` element, but this is meant to contain the title of the resource being quoted, e.g. the name of the book. There is no reason, however, why you couldn't link the text inside `<cite>` to the quote source in some way:

```
<p>
  According to the
  <a href="/en-US/docs/Web/HTML/Element/blockquote"><cite>MDN
blockquote page</cite></a>:
</p><blockquote
  cite="https://developer.mozilla.org/en-US/docs/Web/HTML/Ele
ment/blockquote"><p>
    The <strong>HTML <code>&lt;blockquote&gt;</code> Element
</strong> (or
    <em>HTML Block Quotation Element</em>) indicates that the
enclosed text is
    an extended quotation.
  </p></blockquote><p>
  The quote element — <code>&lt;q&gt;</code> — is
  <q cite="https://developer.mozilla.org/en-US/docs/Web/HTML/
Element/q">
```

```
    intended for short quotations that don't require paragrap
h breaks.
  </q>
  — <a href="/en-US/docs/Web/HTML/Element/q"><cite>MDN q page
</cite></a>.
</p>
```

Abbreviations

Another fairly common element you'll meet when looking around the Web is `<abbr>` — this is used to wrap around an abbreviation or acronym.

Marking up contact details

HTML has an element for marking up contact details — `<address>`. This wraps around your contact details, for example:

```
<address>Chris Mills, Manchester, The Grim North, UK</address
>
```

Superscript and subscript

You will occasionally need to use superscript and subscript when marking up items like dates, chemical formulae, and mathematical equations so they have the correct meaning. The `<sup>` and `<sub>` elements handle this job. For example:

```
<p>My birthday is on the 25<sup>th</sup> of May 2001.</p><p>
  Caffeine's chemical formula is
  C<sub>8</sub>H<sub>10</sub>N<sub>4</sub>O<sub>2</sub>.
</p><p>If x<sup>2</sup> is 9, x must equal 3 or -3.</p>
```

Representing computer code

There are a number of elements available for marking up computer code using HTML:

- `<code>`: For marking up generic pieces of computer code.
- `<pre>`: For retaining whitespace (generally code blocks) — if you use indentation or excess whitespace inside your text, browsers will ignore it and you will not see it on your rendered page. If you wrap the text in `<pre></pre>` tags however, your whitespace will be rendered identically to how you see it in your text editor.
- `<var>`: For specifically marking up variable names.
- `<kbd>`: For marking up keyboard (and other types of) input entered into the computer.
- `<samp>`: For marking up the output of a computer program.

Let's look at examples of these elements and how they're used to represent computer code. If you want to see the full file, take a look at the other-semantics.html sample file. You can download the file and open it in your browser to see for yourself, but here is a snippet of the code:

```
<pre><code>const para = document.querySelector('p');

para.onclick = function() {
  alert('Owww, stop poking me!');
}</code></pre><p>
  You shouldn't use presentational elements like <code>&lt;fo
nt&gt;</code> and
  <code>&lt;center&gt;</code>.
</p><p>
  In the above JavaScript example, <var>para</var> represents
a paragraph
  element.
</p><p>Select all the text with <kbd>Ctrl</kbd>/<kbd>Cmd</kbd
> + <kbd>A</kbd>.</p><pre>$ <kbd>ping mozilla.org</kbd><samp>
PING mozilla.org (63.245.215.20): 56 data bytes
```

```
64 bytes from 63.245.215.20: icmp_seq=0 ttl=40 time=158.233 m
s</samp></pre>
```

Marking up times and dates

HTML also provides the `<time>` element for marking up times and dates in a machine-readable format. For example:

HTMLCopy to Clipboard

```
<time datetime="2016-01-20">20 January 2016</time>
```

Why is this useful? Well, there are many different ways that humans write down dates. The above date could be written as:

- 20 January 2016
- 20th January 2016
- Jan 20 2016
- 20/01/16
- 01/20/16
- The 20th of next month
- 20e Janvier 2016
- 2016 年 1 月 20 日
- And so on

But these different forms cannot be easily recognized by computers — what if you wanted to automatically grab the dates of all events in a page and insert them into a calendar? The `<time>` element allows you to attach an unambiguous, machine-readable time/date for this purpose.

The basic example above just provides a simple machine readable date, but there are many other options that are possible, for example:

HTMLCopy to Clipboard

```
<!-- Standard simple date -->
<time datetime="2016-01-20">20 January 2016</time><!-- Just y
ear and month -->
<time datetime="2016-01">January 2016</time><!-- Just month a
nd day -->
<time datetime="01-20">20 January</time><!-- Just time, hours
and minutes -->
<time datetime="19:30">19:30</time><!-- You can do seconds an
d milliseconds too! -->
<time datetime="19:30:01.856">19:30:01.856</time><!-- Date an
d time -->
<time datetime="2016-01-20T19:30">7.30pm, 20 January 2016</ti
me><!-- Date and time with timezone offset -->
<time datetime="2016-01-20T19:30+01:00">
  7.30pm, 20 January 2016 is 8.30pm in France
</time><!-- Calling out a specific week number -->
<time datetime="2016-W04">The fourth week of 2016</time>
```

Document and website structure

Basic sections of a document

Webpages can and will look pretty different from one another, but they all tend to share similar standard components, unless the page is displaying a fullscreen video or game, is part of some kind of art project, or is just badly structured:

header:Usually a big strip across the top with a big heading, logo, and perhaps a tagline. This usually stays the same from one webpage to another.navigation bar:Links to the site's main sections; usually represented by menu buttons, links, or tabs. Like the header, this content usually remains consistent from one webpage to another — having inconsistent navigation on your website will just lead to confused, frustrated users. Many web designers consider the navigation bar to be part of the header rather than an individual component, but that's not a requirement; in fact, some also argue that having the two separate is better for accessibility, as screen readers can read the two features better if they are separate.main content:A big area in the center that contains most of

the unique content of a given webpage, for example, the video you want to watch, or the main story you're reading, or the map you want to view, or the news headlines, etc. This is the one part of the website that definitely will vary from page to page!sidebar:Some peripheral info, links, quotes, ads, etc. Usually, this is contextual to what is contained in the main content (for example on a news article page, the sidebar might contain the author's bio, or links to related articles) but there are also cases where you'll find some recurring elements like a secondary navigation system.footer:A strip across the bottom of the page that generally contains fine print, copyright notices, or contact info. It's a place to put common information (like the header) but usually, that information is not critical or secondary to the website itself. The footer is also sometimes used for SEO purposes, by providing links for quick access to popular content.

To implement such semantic mark up, HTML provides dedicated tags that you can use to represent such sections, for example:

- **header:** `<header>`.
- **navigation bar:** `<nav>`.
- **main content:** `<main>`, with various content subsections represented by `<article>`, `<section>`, and `<div>` elements.
- **sidebar:** `<aside>`; often placed inside `<main>`.
- **footer:** `<footer>`.

Non-semantic wrappers

Sometimes you'll come across a situation where you can't find an ideal semantic element to group some items together or wrap some content. Sometimes you might want to just group a set of elements together to affect them all as a single entity with some CSS or JavaScript. For cases like these, HTML provides the `<div>` and `<span>` elements. You should use these preferably with a suitable `class` attribute, to provide some kind of label for them so they can be easily targeted.

`<span>` is an inline non-semantic element, which you should only use if you can't think of a better semantic text element to wrap your content, or don't want to add any specific meaning. For example:

```
<p>
  The King walked drunkenly back to his room at 01:00, the be
er doing nothing to
  aid him as he staggered through the door.
  <span class="editor-note">
    [Editor's note: At this point in the play, the lights sho
uld be down low].
  </span></p>
```

In this case, the editor's note is supposed to merely provide extra direction for the director of the play; it is not supposed to have extra semantic meaning. For sighted users, CSS would perhaps be used to distance the note slightly from the main text.

`<div>` is a block level non-semantic element, which you should only use if you can't think of a better semantic block element to use, or don't want to add any specific meaning. For example, imagine a shopping cart widget that you could choose to pull up at any point during your time on an e-commerce site:

HTMLCopy to Clipboard

```
<div class="shopping-cart"><h2>Shopping cart</h2><ul><li><p><
a href=""><strong>Silver earrings</strong></a>: $99.95.
      </p><img src="../products/3333-0985/thumb.png" alt="Sil
ver earrings" /></li><li>…</li></ul><p>Total cost: $237.89</p
></div>
```

This isn't really an `<aside>`, as it doesn't necessarily relate to the main content of the page (you want it viewable from anywhere). It doesn't even particularly warrant using a `<section>`, as it isn't part of the main content of the page. So a `<div>` is fine in this case. We've included a heading as a signpost to aid screen reader users in finding it.

Warning: Divs are so convenient to use that it's easy to use them too much. As they carry no semantic value, they just clutter your HTML code. Take care to use them only when there is no better semantic solution and try to reduce their usage to the minimum otherwise you'll have a hard time updating and maintaining your documents.

Line breaks and horizontal rules

Two elements that you'll use occasionally and will want to know about are `<br>` and `<hr>`.

: the line break element

`<br>` creates a line break in a paragraph; it is the only way to force a rigid structure in a situation where you want a series of fixed short lines, such as in a postal address or a poem. For example:

HTMLPlayCopy to Clipboard

```
<p>
  There once was a man named O'Dell<br />
  Who loved to write HTML<br />
  But his structure was bad, his semantics were sad<br />
  and his markup didn't read very well.
</p>
```

<hr>: the thematic break element

`<hr>` elements create a horizontal rule in the document that denotes a thematic change in the text (such as a change in topic or scene). Visually it just looks like a horizontal line. As an example:

```
<p>
  Ron was backed into a corner by the marauding netherbeasts.
Scared, but
  determined to protect his friends, he raised his wand and p
repared to do
  battle, hoping that his distress call had made it through.
</p><hr /><p>
  Meanwhile, Harry was sitting at home, staring at his royalt
y statement and
  pondering when the next spin off series would come out, whe
n an enchanted
  distress letter flew through his window and landed in his l
ap. He read it
```

```
  hazily and sighed; "better get back to work then", he muse
d.
</p>
```

Planning a simple website

Once you've planned out the structure of a simple webpage, the next logical step is to try to work out what content you want to put on a whole website, what pages you need, and how they should be arranged and link to one another for the best possible user experience. This is called Information architecture. In a large, complex website, a lot of planning can go into this process, but for a simple website of a few pages, this can be fairly simple, and fun!

Images in HTML

There are other types of multimedia to consider, but it is logical to start with the humble `<img>` element used to embed a simple image in a webpage. In this article we'll look at how to use it in more depth, including basics, annotating it with captions using `<figure>`, and how it relates to CSS background images.

Video and audio content

Next, we'll look at how to use the HTML `<video>` and `<audio>` elements to embed video and audio on our pages, including basics, providing access to different file formats to different browsers, adding captions and subtitles, and how to add fallbacks for older browsers.

From <object> to <iframe> — other embedding technologies

At this point we'd like to take somewhat of a sideways step, looking at a couple of elements that allow you to embed a wide variety of content types into your webpages: the `<iframe>`, `<embed>` and `<object>` elements. `<iframe>`s are for embedding other web pages, and the other two allow you to embed external resources such as PDF files.

Adding vector graphics to the Web

Vector graphics can be very useful in certain situations. Unlike regular formats like PNG/JPG, they don't distort/pixelate when zoomed in — they can remain smooth when scaled. This article introduces you to what vector graphics are and how to include the popular SVG format in web pages.

Creating your first table

We've talked table theory enough, so, let's dive into a practical example and build up a simple table.

1. First of all, make a local copy of blank-template.html and minimal-table.css in a new directory on your local machine.
2. The content of every table is enclosed by these two tags: `<table></table>`. Add these inside the body of your HTML.
3. The smallest container inside a table is a table cell, which is created by a `<td>` element ('td' stands for 'table data'). Add the following inside your table tags:

   HTMLCopy to Clipboard

   ```
   <td>Hi, I'm your first cell.</td>
   ```

4. If we want a row of four cells, we need to copy these tags three times. Update the contents of your table to look like so:

   HTMLCopy to Clipboard

   ```
   <td>Hi, I'm your first cell.</td><td>I'm your second cell.
   </td><td>I'm your third cell.</td><td>I'm your fourth cel
   l.</td>
   ```

As you will see, the cells are not placed underneath each other, rather they are automatically aligned with each other on the same row. Each `<td>` element creates a single cell and together they make up the first row. Every cell we add makes the row grow longer.

To stop this row from growing and start placing subsequent cells on a second row, we need to use the `<tr>` element ('tr' stands for 'table row'). Let's investigate this now.

1. Place the four cells you've already created inside `<tr>` tags, like so:

   HTMLCopy to Clipboard

   ```
   <tr><td>Hi, I'm your first cell.</td><td>I'm your second c
   ell.</td><td>I'm your third cell.</td><td>I'm your fourth
   ```

```
cell.</td></tr>
```

2. Now you've made one row, have a go at making one or two more — each row needs to be wrapped in an additional `<tr>` element, with each cell contained in a `<td>`.

CSS

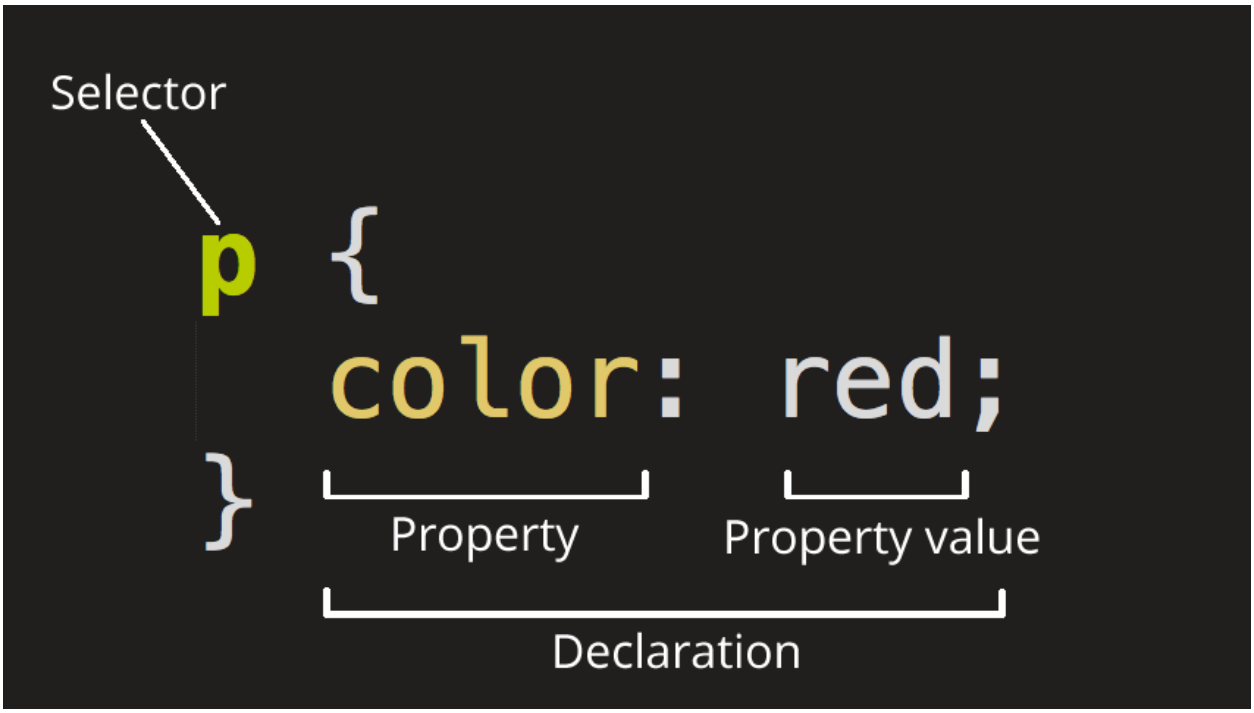
CSS (Cascading Style Sheets) is the code that styles web content. *CSS basics* walks through what you need to get started. We'll answer questions like: How do I make text red? How do I make content display at a certain location in the (webpage) layout? How do I decorate my webpage with background images and colors?

What is CSS?

Like HTML, CSS is not a programming language. It's not a markup language either. **CSS is a style sheet language.** CSS is what you use to selectively style HTML elements. For example, this CSS selects paragraph text, setting the color to red:

Anatomy of a CSS ruleset

Let's dissect the CSS code for red paragraph text to understand how it works:



The whole structure is called a **ruleset**. (The term *ruleset* is often referred to as just *rule*.) Note the names of the individual parts:

SelectorThis is the HTML element name at the start of the ruleset. It defines the element(s) to be styled (in this example, `<p>` elements). To style a different element, change the selector.DeclarationThis is a single rule like `color: red;`. It specifies which of the element's **properties** you want to style.PropertiesThese are ways in which you can style an HTML element. (In this example, `color` is a property of the `<p>` elements.) In CSS, you choose which properties you want to affect in the rule.Property valueTo the right of the property—after the colon—there is the **property value**. This chooses one out of many possible appearances for a given property. (For example, there are many `color` values in addition to `red`.)

Note the other important parts of the syntax:

- Apart from the selector, each ruleset must be wrapped in curly braces. (`{}`)
- Within each declaration, you must use a colon (`:`) to separate the property from its value or values.
- Within each ruleset, you must use a semicolon (`;`) to separate each declaration from the next one.

To modify multiple property values in one ruleset, write them separated by semicolons, like this:

CSSCopy to Clipboard

```
p {
  color: red;
  width: 500px;
  border: 1px solid black;
}
```

Selecting multiple elements

You can also select multiple elements and apply a single ruleset to all of them. Separate multiple selectors by commas. For example:

CSSCopy to Clipboard

```
p,
li,
h1 {
  color: red;
}
```

Different types of selectors

There are many different types of selectors. The examples above use **element selectors**, which select all elements of a given type. But we can make more specific selections as well. Here are some of the more common types of selectors:

Selector name	What does it select	Example
Element selector (sometimes called a tag or type selector)	All HTML elements of the specified type.	`p` selects `<p>`
ID selector	The element on the page with the specified ID. On a given HTML page, each id value should be unique.	`#my-id` selects `<p id="my-id">` or `<a id="my-id">`

Class selector	The element(s) on the page with the specified class. Multiple instances of the same class can appear on a page.	`.my-class` selects `<p class="my-class">` and `<a class="my-class">`
Attribute selector	The element(s) on the page with the specified attribute.	`img[src]` selects `<img src="myimage.png">` but not `<img>`
Pseudo-class selector	The specified element(s), but only when in the specified state. (For example, when a cursor hovers over a link.)	`a:hover` selects `<a>`, but only when the mouse pointer is hovering over the link.

There are many more selectors to discover. To learn more, see the MDN Selectors guide.

Fonts and text

Now that we've explored some CSS fundamentals, let's improve the appearance of the example by adding more rules and information to the `style.css` file.

1. First, find the output from Google Fonts that you previously saved from What will your website look like?. Add the `<link>` element somewhere inside your `index.html`'s head (anywhere between the `<head>` and `</head>` tags). It looks something like this:This code links your page to a style sheet that loads the Open Sans font family with your webpage.

   HTMLCopy to Clipboard

   ```
   <link
     href="https://fonts.googleapis.com/css?family=Open+San
   s"rel="stylesheet" />
   ```

2. Next, delete the existing rule you have in your `style.css` file. It was a good test, but let's not continue with lots of red text.

3. Add the following lines (shown below), replacing the `font-family` assignment with your `font-family` selection from What will your website look like?. The property `font-family` refers to the font(s) you want to use for text. This rule defines a global base font and font size for the whole page. Since `<html>` is the parent

element of the whole page, all elements inside it inherit the same `font-size` and `font-family`.

CSSCopy to Clipboard

```
html {
  font-size: 10px; /* px means "pixels": the base font size is now 10 pixels high */
  font-family: "Open Sans", sans-serif; /* this should be the rest of the output you got from Google Fonts */
}
```

Note: Anything in CSS between `/*` and `*/` is a **CSS comment**. The browser ignores comments as it renders the code. CSS comments are a way for you to write helpful notes about your code or logic.

4. Now let's set font sizes for elements that will have text inside the HTML body (<h1>, `<li>`, and `<p>`). We'll also center the heading. Finally, let's expand the second ruleset (below) with settings for line height and letter spacing to make body content more readable.

   CSSCopy to Clipboard

```
h1 {
  font-size: 60px;
  text-align: center;
}

p,
li {
  font-size: 16px;
  line-height: 2;
  letter-spacing: 1px;
}
```

CSS: all about boxes

Something you'll notice about writing CSS: a lot of it is about boxes. This includes setting size, color, and position. Most HTML elements on your page can be thought of as boxes sitting on top of other boxes.

CSS layout is mostly based on the *box model.* Each box taking up space on your page has properties like:

- `padding`, the space around the content. In the example below, it is the space around the paragraph text.
- `border`, the solid line that is just outside the padding.
- `margin`, the space around the outside of the border.

In this section we also use:

- `width` (of an element).
- `background-color`, the color behind an element's content and padding.
- `color`, the color of an element's content (usually text).
- `text-shadow` sets a drop shadow on the text inside an element.
- `display` sets the display mode of an element. (keep reading to learn more)

To continue, let's add more CSS. Keep adding these new rules at the bottom of `style.css`. Experiment with changing values to see what happens.

Changing the page color

CSSCopy to Clipboard

```
html {
  background-color: #00539f;
}
```

This rule sets a background color for the entire page. Change the color code to the color you chose in What will my website look like?.

Styling the body

CSSCopy to Clipboard

```css
body {
  width: 600px;
  margin: 0 auto;
  background-color: #ff9500;
  padding: 0 20px 20px 20px;
  border: 5px solid black;
}
```

There are several declarations for the `<body>` element. Let's go through these line-by-line:

- `width: 600px;` This forces the body to always be 600 pixels wide.
- `margin: 0 auto;` When you set two values on a property like `margin` or `padding`, the first value affects the element's top *and* bottom side (setting it to `0` in this case); the second value affects the left *and* right side. (Here, `auto` is a special value that divides the available horizontal space evenly between left and right). You can also use one, two, three, or four values, as documented in Margin Syntax.
- `background-color: #FF9500;` This sets the element's background color. This project uses a reddish orange for the body background color, as opposed to dark blue for the `<html>` element. (Feel free to experiment.)
- `padding: 0 20px 20px 20px;` This sets four values for padding. The goal is to put some space around the content. In this example, there is no padding on the top of the body, and 20 pixels on the right, bottom and left. The values set top, right, bottom, left, in that order. As with `margin`, you can use one, two, three, or four values, as documented in Padding Syntax.
- `border: 5px solid black;` This sets values for the width, style and color of the border. In this case, it's a five-pixel–wide, solid black border, on all sides of the body.

Positioning and styling the main page title

CSSCopy to Clipboard

```css
h1 {
  margin: 0;
  padding: 20px 0;
  color: #00539f;
  text-shadow: 3px 3px 1px black;
}
```

You may have noticed there's a horrible gap at the top of the body. That happens because browsers apply default styling to the h1 element (among others). That might seem like a bad idea, but the intent is to provide basic readability for unstyled pages. To eliminate the gap, we overwrite the browser's default styling with the setting `margin: 0;`.

Next, we set the heading's top and bottom padding to 20 pixels.

Following that, we set the heading text to be the same color as the HTML background color.

Finally, `text-shadow` applies a shadow to the text content of the element. Its four values are:

- The first pixel value sets the **horizontal offset** of the shadow from the text: how far it moves across.
- The second pixel value sets the **vertical offset** of the shadow from the text: how far it moves down.
- The third pixel value sets the **blur radius** of the shadow. A larger value produces a more fuzzy-looking shadow.
- The fourth value sets the base color of the shadow.

Try experimenting with different values to see how it changes the appearance.

Centering the image

CSSCopy to Clipboard

```css
img {
  display: block;
```

```
  margin: 0 auto;
}
```

Next, we center the image to make it look better. We could use the `margin: 0 auto` trick again as we did for the body. But there are differences that require an additional setting to make the CSS work.

The `<body>` is a **block** element, meaning it takes up space on the page. The margin applied to a block element will be respected by other elements on the page. In contrast, images are **inline** elements, for the auto margin trick to work on this image, we must give it block-level behavior using `display: block;`.

Note: The instructions above assume that you're using an image smaller than the width set on the body. (600 pixels) If your image is larger, it will overflow the body, spilling into the rest of the page. To fix this, you can either: 1) reduce the image width using a graphics editor, or 2) use CSS to size the image by setting the `width` property on the `<img>` element with a smaller value.

Applying CSS to HTML

First, let's examine three methods of applying CSS to a document: with an external stylesheet, with an internal stylesheet, and with inline styles.

External stylesheet

An external stylesheet contains CSS in a separate file with a `.css` extension. This is the most common and useful method of bringing CSS to a document. You can link a single CSS file to multiple web pages, styling all of them with the same CSS stylesheet. In the Getting started with CSS, we linked an external stylesheet to our web page.

You reference an external CSS stylesheet from an HTML `<link>` element:

HTMLCopy to Clipboard

```
<!doctype html><html lang="en-GB"><head><meta charset="utf
-8" /><title>My CSS experiment</title><link rel="styleshee
t" href="styles.css" /></head><body><h1>Hello World!</h1><
p>This is my first CSS example</p></body></html>
```

The CSS stylesheet file might look like this:

CSSCopy to Clipboard

```css
h1 {
  color: blue;
  background-color: yellow;
  border: 1px solid black;
}

p {
  color: red;
}
```

The `href` attribute of the `<link>` element needs to reference a file on your file system. In the example above, the CSS file is in the same folder as the HTML document, but you could place it somewhere else and adjust the path. Here are three examples:

HTMLCopy to Clipboard

```html
<!-- Inside a subdirectory called styles inside the curren
t directory -->
<link rel="stylesheet" href="styles/style.css" /><!-- Insi
de a subdirectory called general, which is in a subdirecto
ry called styles, inside the current directory -->
<link rel="stylesheet" href="styles/general/style.css" />
<!-- Go up one directory level, then inside a subdirectory
called styles -->
<link rel="stylesheet" href="../styles/style.css" />
```

Internal stylesheet

An internal stylesheet resides within an HTML document. To create an internal stylesheet, you place CSS inside a `<style>` element contained inside the HTML `<head>`.

The HTML for an internal stylesheet might look like this:

HTMLCopy to Clipboard

```
<!doctype html><html lang="en-GB"><head><meta charset="utf
-8" /><title>My CSS experiment</title><style>
      h1 {
        color: blue;
        background-color: yellow;
        border: 1px solid black;
      }

      p {
        color: red;
      }
    </style></head><body><h1>Hello World!</h1><p>This is m
y first CSS example</p></body></html>
```

In some circumstances, internal stylesheets can be useful. For example, perhaps you're working with a content management system where you are blocked from modifying external CSS files.

But for sites with more than one page, an internal stylesheet becomes a less efficient way of working. To apply uniform CSS styling to multiple pages using internal stylesheets, you must have an internal stylesheet in every web page that will use the styling. The efficiency penalty carries over to site maintenance too. With CSS in internal stylesheets, there is the risk that even one simple styling change may require edits to multiple web pages.

Inline styles

Inline styles are CSS declarations that affect a single HTML element, contained within a `style` attribute. The implementation of an inline style in an HTML document might look like this:

HTMLCopy to Clipboard

```
<!doctype html><html lang="en-GB"><head><meta charset="utf
-8" /><title>My CSS experiment</title></head><body><h1 sty
le="color: blue;background-color: yellow;border: 1px solid
```

```
black;">
      Hello World!
    </h1><p style="color:red;">This is my first CSS exampl
e</p></body></html>
```

Avoid using CSS in this way, when possible. It is the opposite of a best practice. First, it is the least efficient implementation of CSS for maintenance. One styling change might require multiple edits within a single web page. Second, inline CSS also mixes (CSS) presentational code with HTML and content, making everything more difficult to read and understand. Separating code and content makes maintenance easier for all who work on the website.

There are a few circumstances where inline styles are more common. You might have to resort to using inline styles if your working environment is very restrictive. For example, perhaps your CMS only allows you to edit the HTML body. You may also see a lot of inline styles in HTML email to achieve compatibility with as many email clients as possible.

Playing with the CSS in this article

For the exercise that follows, create a folder on your computer. You can name the folder whatever you want. Inside the folder, copy the text below to create two files:

index.html:

HTMLCopy to Clipboard

```
<!doctype html><html lang="en"><head><meta charset="utf-8"
/><meta name="viewport" content="width=device-width" /><ti
tle>My CSS experiments</title><link rel="stylesheet" href
="styles.css" /></head><body><p>Create your test HTML here
</p></body></html>
```

styles.css:

CSSCopy to Clipboard

```
/* Create your test CSS here */
```

```
p {
  color: red;
}
```

When you find CSS that you want to experiment with, replace the HTML `<body>` contents with some HTML to style, and then add your test CSS code to your CSS file.

Selectors

A selector targets HTML to apply styles to content. We have already discovered a variety of selectors in the Getting started with CSS tutorial. If CSS is not applying to content as expected, your selector may not match the way you think it should match.

Each CSS rule starts with a selector — or a list of selectors — in order to tell the browser which element or elements the rules should apply to. All the examples below are valid selectors or lists of selectors.

CSSCopy to Clipboard

```
h1
a:link
.manythings
#onething
*
.box p
.box p:first-child
h1, h2, .intro
```

Try creating some CSS rules that use the selectors above. Add HTML to be styled by the selectors. If any of the syntax above is not familiar, try searching MDN.

Note: You will learn more about selectors in the next module: CSS selectors.

Specificity

You may encounter scenarios where two selectors select the same HTML element. Consider the stylesheet below, with a `p` selector that sets paragraph text to blue.

However, there is also a class that sets the text of selected elements to red.

CSSCopy to Clipboard

```css
.special {
  color: red;
}

p {
  color: blue;
}
```

Suppose that in our HTML document, we have a paragraph with a class of `special`. Both rules apply. Which selector prevails? Do you expect to see blue or red paragraph text?

HTMLCopy to Clipboard

```html
<p class="special">What color am I?</p>
```

The CSS language has rules to control which selector is stronger in the event of a conflict. These rules are called **cascade** and **specificity**. In the code block below, we define two rules for the `p` selector, but the paragraph text will be blue. This is because the declaration that sets the paragraph text to blue appears later in the stylesheet. Later styles replace conflicting styles that appear earlier in the stylesheet. This is the **cascade** rule.

CSSCopy to Clipboard

```css
p {
  color: red;
}

p {
  color: blue;
}
```

However, in the case of our earlier example with the conflict between the class selector and the element selector, the class prevails, rendering the paragraph text red. How can this happen even though a conflicting style appears later in the stylesheet? A class is rated as being more specific, as in having more **specificity** than the element selector, so it cancels the other conflicting style declaration.

Try this experiment for yourself! Add HTML, then add the two `p { }` rules to your stylesheet. Next, change the first `p` selector to `.special` to see how it changes the styling.

The rules of specificity and the cascade can seem complicated at first. These rules are easier to understand as you become more familiar with CSS. The Cascade and inheritance section in the next module explains this in detail, including how to calculate specificity.

For now, remember that specificity exists. Sometimes, CSS might not apply as you expected because something else in the stylesheet has more specificity. Recognizing that more than one rule could apply to an element is the first step in fixing these kinds of issues.

Properties and values

At its most basic level, CSS consists of two components:

- **Properties**: These are human-readable identifiers that indicate which stylistic features you want to modify. For example, `font-size`, `width`, `background-color`.
- **Values**: Each property is assigned a value. This value indicates how to style the property.

Setting CSS properties to specific values is the primary way of defining layout and styling for a document. The CSS engine calculates which declarations apply to every single element of a page.

CSS properties and values are case-insensitive. The property and value in a property-value pair are separated by a colon (`:`).

Look up different values of properties listed below. Write CSS rules that apply styling to different HTML elements:

- `font-size`

- `width`
- `background-color`
- `color`
- `border`

Warning: If a property is unknown, or if a value is not valid for a given property, the declaration is processed as *invalid*. It is completely ignored by the browser's CSS engine.

Warning: In CSS (and other web standards), it has been agreed that US spelling is the standard where there is language variation or uncertainty. For example, `colour` should be spelled `color`, as `colour` will not work.

Functions

While most values are relatively simple keywords or numeric values, there are some values that take the form of a function.

The calc() function

An example would be the `calc()` function, which can do simple math within CSS:

HTMLPlayCopy to Clipboard

```
<div class="outer"><div class="box">The inner box is 90% -
30px.</div></div>
```

CSSPlayCopy to Clipboard

```
.outer {
  border: 5px solid black;
}

.box {
  padding: 10px;
  width: calc(90% - 30px);
  background-color: rebeccapurple;
```

```
  color: white;
}
```

A function consists of the function name, and parentheses to enclose the values for the function. In the case of the `calc()` example above, the values define the width of this box to be 90% of the containing block width, minus 30 pixels. The result of the calculation isn't something that can be computed in advance and entered as a static value.

Transform functions

Another example would be the various values for `transform`, such as `rotate()`.

HTMLPlayCopy to Clipboard

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

CSSPlayCopy to Clipboard

```
.box {
  margin: 30px;
  width: 100px;
  height: 100px;
  background-color: rebeccapurple;
  transform: rotate(0.8turn);
}
```

Look up different values of properties listed below. Write CSS rules that apply styling to different HTML elements:

- `transform`
- `background-image`**, in particular gradient values**
- `color`**, in particular rgb and hsl values**

@rules

CSS @rules (pronounced "at-rules") provide instruction for what CSS should perform or how it should behave. Some @rules are simple with just a keyword and a value. For example, `@import` imports a stylesheet into another CSS stylesheet:

CSSCopy to Clipboard

```
@import "styles2.css";
```

One common @rule that you are likely to encounter is `@media`, which is used to create media queries. Media queries use conditional logic for applying CSS styling.

In the example below, the stylesheet defines a default pink background for the `<body>` element. However, a media query follows that defines a blue background if the browser viewport is wider than 30em.

CSSCopy to Clipboard

```
body {
  background-color: pink;
}

@media (min-width: 30em) {
  body {
    background-color: blue;
  }
}
```

You will encounter other @rules throughout these tutorials.

See if you can add a media query that changes styles based on the viewport width. Change the width of your browser window to see the result.

Shorthands

Some properties like `font`, `background`, `padding`, `border`, and `margin` are called **shorthand properties.** This is because shorthand properties set several values in a single line.

For example, this one line of code:

CSSCopy to Clipboard

```
/* In 4-value shorthands like padding and margin, the valu
es are applied
   in the order top, right, bottom, left (clockwise from t
he top). There are also other
   shorthand types, for example 2-value shorthands, which
set padding/margin
   for top/bottom, then left/right */
padding: 10px 15px 15px 5px;
```

is equivalent to these four lines of code:

CSSCopy to Clipboard

```
padding-top: 10px;
padding-right: 15px;
padding-bottom: 15px;
padding-left: 5px;
```

This one line:

CSSCopy to Clipboard

```
background: red url(bg-graphic.png) 10px 10px repeat-x fix
ed;
```

is equivalent to these five lines:

CSSCopy to Clipboard

```
background-color: red;
background-image: url(bg-graphic.png);
background-position: 10px 10px;
background-repeat: repeat-x;
background-attachment: fixed;
```

Later in the course, you will encounter many other examples of shorthand properties. MDN's CSS reference is a good resource for more information about any shorthand property.

Try using the declarations (above) in your own CSS exercise to become more familiar with how it works. You can also experiment with different values.

Warning: One less obvious aspect of using CSS shorthand is how omitted values reset. A value not specified in CSS shorthand reverts to its initial value. This means an omission in CSS shorthand can **override previously set values**.

Comments

As with any coding work, it is best practice to write comments along with CSS. This helps you to remember how the code works as you come back later for fixes or enhancement. It also helps others understand the code.

CSS comments begin with `/*` and end with `*/`. In the example below, comments mark the start of distinct sections of code. This helps to navigate the codebase as it gets larger. With this kind of commenting in place, searching for comments in your code editor becomes a way to efficiently find a section of code.

CSSCopy to Clipboard

```
/* Handle basic element styling */
/* ------------------------------------------------------------
------------------------------------ */
body {
  font:
    1em/150% Helvetica,
    Arial,
    sans-serif;
  padding: 1em;
  margin: 0 auto;
  max-width: 33em;
}

@media (min-width: 70em) {
  /* Increase the global font size on larger screens or wi
```

```
ndows
     for better readability */
  body {
    font-size: 130%;
  }
}

h1 {
  font-size: 1.5em;
}

/* Handle specific elements nested in the DOM */
div p,
#id:first-line {
  background-color: red;
  border-radius: 3px;
}

div p {
  margin: 0;
  padding: 1em;
}

div p + p {
  padding-top: 0;
}
```

"Commenting out" code is also useful for temporarily disabling sections of code for testing. In the example below, the rules for `.special` are disabled by "commenting out" the code.

CSSCopy to Clipboard

```
/*.special {
  color: red;
}*/
```

```
p {
  color: blue;
}
```

Add comments to your CSS.

White space

White space means actual spaces, tabs and new lines. Just as browsers ignore white space in HTML, browsers ignore white space inside CSS. The value of white space is how it can improve readability.

In the example below, each declaration (and rule start/end) has its own line. This is arguably a good way to write CSS. It makes it easier to maintain and understand CSS.

CSSCopy to Clipboard

```
body {
  font:
    1em/150% Helvetica,
    Arial,
    sans-serif;
  padding: 1em;
  margin: 0 auto;
  max-width: 33em;
}

@media (min-width: 70em) {
  body {
    font-size: 130%;
  }
}

h1 {
  font-size: 1.5em;
```

```
}

div p,
#id:first-line {
  background-color: red;
  border-radius: 3px;
}

div p {
  margin: 0;
  padding: 1em;
}

div p + p {
  padding-top: 0;
}
```

The next example shows the equivalent CSS in a more compressed format. Although the two examples work the same, the one below is more difficult to read.

CSSCopy to Clipboard

```
body {font: 1em/150% Helvetica, Arial, sans-serif; paddin
g: 1em; margin: 0 auto; max-width: 33em;}
@media (min-width: 70em) { body { font-size: 130%;}}

h1 {font-size: 1.5em;}

div p, #id:first-line {background-color: red; border-radiu
s: 3px;}
div p {margin: 0; padding: 1em;}
div p + p {padding-top: 0;}
```

For your own projects, you will format your code according to personal preference. For team projects, you may find that a team or project has its own style guide.

Warning: Though white space separates values in CSS declarations, **property names never have white space**.

For example, these declarations are valid CSS:

CSSCopy to Clipboard

```
margin: 0 auto;
padding-left: 10px;
```

But these declarations are invalid:

CSSCopy to Clipboard

```
margin: 0auto;
padding- left: 10px;
```

Do you see the spacing errors? First, `0auto` is not recognized as a valid value for the `margin` property. The entry `0auto` is meant to be two separate values: `0` and `auto`. Second, the browser does not recognize `padding-` as a valid property. The correct property name (`padding-left`) is separated by an errant space.

You should always make sure to separate distinct values from one another by at least one space. Keep property names and property values together as single unbroken strings.