Client-Side Web Development

Joel Ross and Mike Freeman

August 29, 2017

Contents

About this Book

This book covers the the skills and techniques necessary for creating sophisticated and accessible interactive web applications. It focuses on the client-side languages, tools, and libraries that professionals use to build the web sites you use every day. It assumes a basic background in computer programming (e.g., one course in Java, and some concepts from the technical foundations of informatics). These materials were developed for the **INFO 343: Client-Side Web Development** course taught at the University of Washington Information School; however they have been structured to be an online resource for anyone who wishes to learn modern web programming techniques.

Some content has been adapted from tutorials by David Stearns.

This book is currently in **alpha** status. Visit us on GitHub to contribute improvements.

Chapter 1

Getting Setup

This course will cover a wide variety of tools and techniques used in modern web development, including different software programs that are used to write, manage, and execute the code for your web application. This chapter explains how to install and use some of the software you will need to utilize.

Note that iSchool lab machines should have all appropriate software already installed and ready to use.

1.1 Web Browser

The first thing you'll need is a web browser for viewing the web pages you make! We recommend you install and utilize **Chrome**, which comes with an effective set of built-in developer tools that will be especially useful in this class.

- You can access the Chrome Developer tools by selecting **View > Developer > Developer Tools** from Chrome's main menu (`Cmd+Option+I` on a Mac, `Ctrl+Shift+I` on Windows). You will almost always want to have these tools open when doing web development, particularly when including interactivity via JavaScript.

Other modern browsers such as **Firefox** or **Microsoft Edge** will also function in this class and include their own versions of the required development tools. Note that different browsers may and will render code in different ways which will be discussed extensively throughout the course.

1.2 Code Editors

In order to write web code, you need somewhere to write it. There are a variety of code editors and IDEs (Integrated Development Environments) that are specialized for web development, providing syntax highlighting, code completion, and other useful functionality. There are lots of different code editors out there, all of which have slightly different appearances and features. You only need to download and use one of the following programs (we recommend **Visual Studio Code** as a default), but feel free to try out different ones to find something you like (and then evangelize about it to your friends!)

Visual Studio Code

Visual Studio Code (or VS Code; not to be confused with Visual Studio) is a free, open-source editor developed by Microsoft—yes, really. It focuses on web programming and JavaScript, though also supports many other languages and provides a number of community-built extensions for adding even more features. Although fairly new, it is updated regularly and has has become my main editors for programming. VS Code is actually a stand-alone web application, so it's written in the same HTML, CSS, and JavaScript you'll learn in this course!

To install VS Code, follow the above link and Click the "Download" button to download the installer (e.g, `.exe`) file, then double-click on that to install the application.

Once you've installed VS Code, the trick to using it effectively is to get comfortable with the Command Palette. If you hit `Cmd+Shift+P`, VS Code will open a small window where you can search for whatever you want the editor to do. For example, if you type in `markdown` you can get list of commands related to Markdown files (including the ability to open up a preview). The `Format Code` option is particularly useful.

For more information about using VS Code, see the documentation, which includes videos if you find them useful. The documentation for programming in HTML, CSS, and especially JavaScript also contain lots of tips and tricks.

Atom

Atom is a text editor built by the folks at GitHub and has been gaining in popularity. It is very similar to VS Code in terms of features, but has a somewhat different interface and community. It has a similar *command-palette* to VS Code, and is arguably even nicer about editing Markdown specifically. The document you are reading was authored in Atom.

Brackets

Brackets is a coding editor authored by Adobe specialized for client-side web developers. It has some intriguing features that are not yet in Visual Studio Code, as well as possibly the nicest interface of this list.

Sublime Text

Sublime Text is a very popular text editor with excellent defaults and a variety of available extensions (though you'll need to manage and install extensions to achieve the functionality offered by other editors out of the box). While the software can be used for free, every 20 or so saves it will prompt you to purchase the full version. This is my application of choice for when I just write to write a plain text file.

1.3 Bash (Command Line)

Many of the software tools used in professional web development are used on the **command-line**: a text-based interface for controlling your computer. While the command-line is harder to learn and figure out, it is particularly effective for doing web development. Command-line automation is powerful and efficient enough to handle the dozens of repeated tasks across hundreds of different source files (split across multiple computers) commonly found in web programming. You will need to be comfortable using the command-line in order to utilize the software for this course.

While there are multiple different **command shells** (command line interfaces), this course is based on the Bash shell, which provides a particular common set of commands common to Mac and Linux machines.

On a Mac you'll want to use the built-in app called **Terminal**. You can open it by searching via Spotlight (hit Cmd (⌘) and Spacebar together, type in "terminal", then select the app to open it), or by finding it in the Applications/Utilities folder.

On Windows, we recommend using **Git Bash**, which you should install along with `git` (see below). Open this program to open the command-shell.

- Note that Windows does come with its own command-prompt, called the *DOS Prompt*, but it has a different set of commands and features. *Powershell* is a more powerful version of the DOS prompt if you really want to get into the Windows Management Framework. But Bash is more common in open-source programming like we'll be doing, and so we will be focusing on that set of commands.

- Alternatively, the 64-bit Windows 10 Anniversary update (August 2016) *does* include a beta version of an integrated Bash shell. You can access this by enabling the subsystem for Linux and then running `bash` in the command prompt. This is currently (May 2017) "beta" technology, but will suffice for our purposes if you can get it running.

This course expects you to already be familiar with basic command-line usage. For review, see The Command Line in the *INFO 201* course reader.

1.4 Git and GitHub

Professional web development involves many different people working on many different files. **`git`** is a collaborative version control system that provides a set of commands that allow you to manage changes to written code, particularly when collaborating with other programmers.

You will need to download and install the software. If you are on a Mac, `git` should already be installed. If you are using a Windows machine, then installing `git` will also install Git Bash, a command shell (described avove).

Note that `git` is a command-line application: you can test that it is installed by running the command:

```
git --version
```

While `git` is the software used to manage versions of code, **GitHub** is a website that is used to store copies of computer code that are being managed with `git` (think "Imgur for code").

In order to use GitHub, you'll need to create a free GitHub account, if you don't already have one. You should register a username that is identifiable as you (e.g., based on your name or your UW NetID). This will make it easier for others to determine out who contributed what code, rather than needing to figure out who 'LeetDesigner2099' is. This can be the start of a professional account you may use for the rest of your career!

- Note that you can have `git` save your GitHub password on your local machine so you don't have to type it repeatedly. See Authenticating with GitHub from Git.

This course expects you to already be familiar with utilizing Git and GitHub. For review, see Git and GitHub and Git Branches and Collaboration in the *INFO 201* course reader. Note that students in the INFO 343 course will be using GitHub and Pull Requests to turn in programming assignments.

1.5 Node and npm

Node.js (commonly just "Node") is a a command-line runtime environment for the JavaScript programming language—that is, a program that is used to *interpret* and *execute* programming instructions written in JavaScript. Although client-side development usually involves running JavaScript in the browser (see Chapter: JavaScript), Node provides a platform for installing and running a wide variety of "helper" programs that are frequently used in web development.

To install Node, visit the download page and select the installer for your operating system (you probably want the `.msi` for Windows and the `.pkg` for Mac). For this course you will want to install the **latest version** of Node (6.10+), so you should update it if you haven't in a while. Node is a command-line application, so you can test that it is installed and available to your command shell (e.g., Terminal or Git Bash) with:

```
node --version
```

Installing Node also installs an additional command-line program called ***npm***. *npm* is a **package manager**, or a program used to "manage" other programs—think of it as a command-line "app store" for developer tools and libraries. *npm* is the most common way of installing and running a large number of tools used in professional web development. At the time of writing, the *npm* "registry" lists around 500,000 different packages.

Managing packages with npm

You can use the `npm` program to download and install command-line programs by name:

```
npm install -g PACKAGE-NAME
```

For example, you can install the *live-server* utility (a simple program that runs a local web server and will automatically "refresh" the browser when your code changes) using

```
npm install -g live-server
```

- Once the program is done installing, you can run it from the command-line by using the command `live-server`. This program will serve all of the content from the current directory. See Chapter 2 for details.

Importantly, note the included **`-g`** option. This tells *npm* that the package should be installed **globally**, making it available across the entire computer, rather than just from a particular folder. Because you want to be able to use a command-line utility like *live-server* from any folder (e.g., for any project), command-line utilities are always installed globally with the `-g` option.

It is also possible to omit that option and install a package *locally*. For example:

```
npm install lodash
```

Will download the `lodash` code library (a set of useful JavaScript functions). This package will be placed into a new folder *in the current project directory* called `node_modules/`, and can be imported and used in the current directory's code. (It's called a local install because the package is only available to the "local" project). You will of course need to install local packages once per project.

Because node packages can be very large, and projects can have lots of them, you want to be sure to **not** commit the `node_modules/` folder to version control. Make sure that the folder is listed in your `.gitignore` file!

1.5.1 `package.json`

As projects become large, it is common for them to build up many *dependencies*: packages that must be installed in order for the program to work. In other words, there needs to be a certain set of packages in the project's `node_modules/` folder. *npm* is able to keep track of these dependencies by recording them in a specialized file called `package.json` that can be placed inside the project directory. A `package.json` file is a text file containing a JSON list of information about your project. For example:

```
{
  "name": "example",
  "version": "1.0.0",
  "private": true,
  "description": "A project with an example package.json",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "author": "Joel Ross",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.4",
    "moment": "^2.18.1"
    },
  "devDependencies": {
    "html-validator": "^2.2.2"
  }
}
```

(You can create one of these files by using the command `npm init` in the current project directory, and then following the instructions to fill in the fields).

Notice that there are two packages listed under `"dependencies"`: `lodash` and `moment` (the `^4.17.4` indicates which version of `lodash`). You can use `npm` to automatically install all of packages listed under `"dependencies"` (as well as `"devDependencies"`) using the command:

```
npm install
```

Thus using `npm install` without any arguments means "install all of the requirements that have been listed for this project". This is a good first step *any time* you download a project or checkout a repository from GitHub.

When installing specific packages, you can have *npm* add them to the dependencies list by using the `--save` option:

```
npm install --save lodash
```

will install `lodash` locally, and list it in the `package.json` file as a dependency.

Similarly, the `--save-dev` option will instead save the package in the `"devDependencies"` list, which are dependencies needed only for development (writing the program's code) and not for execution (running the program).

You can uninstall packages using `npm uninstall`, or can remove packages from the dependencies lists simply by editing the `package.json` file (e.g., with VS Code).

To sum up, you will use three commands with `npm` to install packages:

1. `npm install -g PACKAGE-NAME` to *globally* install command-line programs
2. `npm install` to *locally* install all of the dependencies for a project you check out
3. `npm install --save PACKAGE-NAME` to *locally* install a new code package and record it in the `package.json` file.

While *npm* is the most popular package manager (and the one utilized in this course), there are others as well. For example, **Yarn** is a package manager created by Facebook that is compatible with *npm* and is quickly growing in popularity.

Resources

Links to the recommended software are collected here for easy access:

- Chrome
- git (and Git Bash) - GitHub (sign up)
  - optional: Bash on Windows
- Visual Studio Code
- Node.js (and npm)

- npm documentation

Chapter 2

Client-Side Development

Web development is the process of implementing (programming) web sites and applications that users can access over the internet. However, the internet is a network involving *many* different computers all communicating with one another. These computers can be divided into two different groups: **servers** store ("host") content and provide ("serve") it to other computers, while **clients** request that content and then present it to the human users.

Consider the process of viewing a basic web page, such as the Wikipedia entry on Informatics. In order to visit this page, the user types the web address (`https://en.wikipedia.org/wiki/Informatics`) into the URL bar, or clicks on a link to go to the page. In either case, user's computer is the **client**, and their browser takes that address or link and uses it to create an **HTTP Request**—a *request* for data sent following the ***H**yper**T**ext **T**ransfer **P**rotocol*. This request is like a letter asking for information, and is sent to a different computer: the **web server** that contains that information.

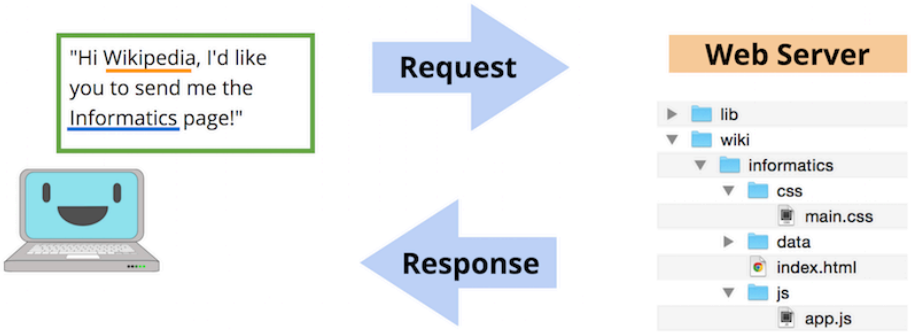

Figure 2.1: A diagram of client/server communication.

The web server will receive this request, and based on its content (e.g., where it was sent to) will decide what information to send as a **response** to the client. In general, this response will be made up of lots of different files: the text content of the web page, styling information (font, color) for how it should look, instructions for respond to user interaction (button clicks), images or other assets to show, and so forth.

The client's web browser will then take all of these different files in the response and use them to *render* the web page for the user to see: it wil determine what text to show, what font and color to make that text, where to put the images, and is ready to do something else when the user clicks on one of those images. Indeed, a web browser is just a computer program that is able to send HTTP requests on behalf of the user, and then render the resulting response.

Given this interaction, **client-side web development** involves implementing programs (writing code) that is interpreted by the *browser*, and so is executed by the client. It is authoring the code that is sent in the server's response. This code specifies how websites should appear and how the user should interact with them. On the other hand, **server-side web development** involves implementing programs that the *server* uses to determine which client-side code is delivered. As an example, a server-side program contains the logic to determine which cat picture should be sent along with the request, while a client-side program contains the logic about where and how that picture should appear on the page.

This course focuses on *client-side web development*, or developing programs that are executed by the browser (generally as a response to a web server request). While we will cover how client-side programs can interact with a server, many of the concepts discussed here can also be run inside a browser without relying on an external server (called "running locally").

2.1 Client-Side File Types

It is the web browser's job to interpret and render the source code files sent by a server as part of an HTTP response. As a client-side web programmer, your task is to write this source code for the browser to interpret. There are multiple different types of source code files, including:

- **`.html`** files containing code written in HTML (HyperText Markup Language). This code will specify the textual and *semantic* content of the web page. See the chapter HTML Fundamentals for details on HTML.
- **`.css`** files containing code written in CSS (Cascading Style Sheets). This code is used to specify styling and *visual appearance* properties (e.g., color and font) for the HTML content. See the chapter CSS Fundamentals for details on CSS.

- **`.js`** files containing code written in JavaScript. This code is used to specify *interactive behaviors* that the website will perform—for example, what should change when the user clicks a button. Note that JavaScript code are "programs" that sent over by the web server as part of the response, but are *executed* on the client's computer. See the chapter JavaScript Fundamentals for details on JavaScript.

HTTP responses may also include additional **asset** files, such as images (`.png`, `.jpg`, `.gif`, etc), fonts, video or music files, etc.

2.2 HTTP Requests and Servers

Modern web browsers are able to *render* (interpret and display) all of these types of files, combining them together into the modern, interactive web pages you use every day. In fact, you can open up almost any file inside a web browser, such as by right-clicking on the file and selecting "Open With", or dragging the file into the browser program. HTML files act as the basis for web pages, so you can open a `.html` file inside your web browser by double-clicking on it (the same way you would open a `.docx` file in MS Word):

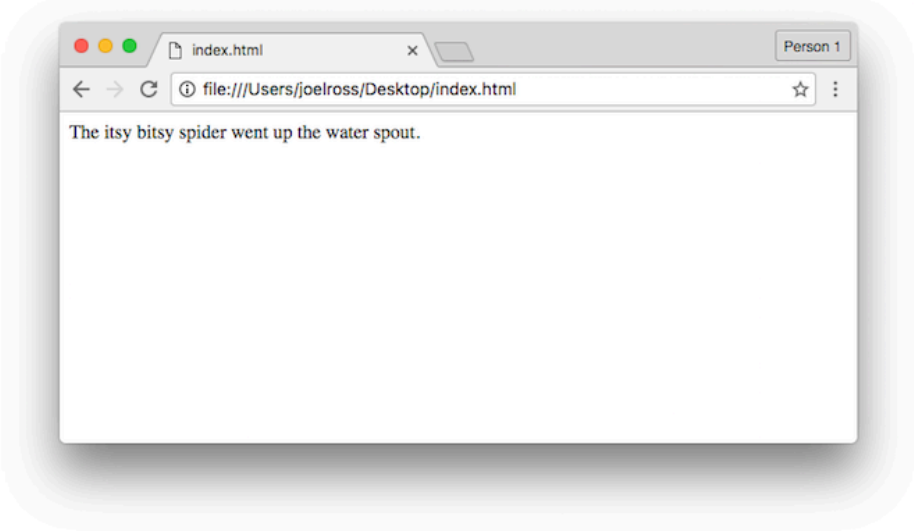

Figure 2.2: An very simple HTML file. See Chapter 3 for source code.

Consider the URL bar in the above browser. The URL (Uniform Resource Locator) is actually a specialized version of a **URI (Uniform Resource Identifier)**. URIs act a lot like the *address* on a postal letter sent within a large organization such as a university: you indicate the business address as well as the department

and the person, and will get a different response (and different data) from Alice in Accounting than from Sally in Sales.

- Note that the URI is the **identifier** (think: variable name) for the resource, while the **resource** is the actual *data* value (the file) that you want to access.

Like postal letter addresses, URIs have a very specific format used to direct the request to the right resource.

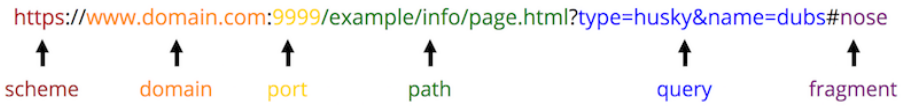

Figure 2.3: The format (schema) of a URI.

The parts of this URI format include:

- **`scheme`** (also **`protocol`**): the "language" that the computer will use to send the request for the resource (file).

  In the example browser window above, the protocol is `file`, meaning that the computer is accessing the resource from the file system. When sending requests to web servers, you would use `https` (**s**ecure HTTP). *Don't use insecure* `http`*!*

  Web page *hyperlinks* often include URIs with the `mailto` protocol (for email links) or the `tel` for phone numbers.

- **`domain`**: the address of the web server to request information from. You can think of this as the recipient of the request letter.

  In the browser window example, there is no domain because the `file` protocol doesn't require it.

- **`port`** (*optional*): used to determine where to connect to the web server. By default, web requests use port 80, but some web servers accept connections on other ports—e.g., `8080`, `8000` and `3000` are all common on development servers, described below.

- **`path`**: which resource on that web server you wish to access. For the `file` protocol, this is the *absolute path* to the file. But even when using `https`, for many web servers, this will be the *relative path* to the file, starting from the "root" folder of that server (which may not be the computer's root folder). For example, if a server used `/Users/joelross/` as its root, then the *path* to the above HTML file would be `Desktop/index.html` (e.g., `https://domain/Desktop/index.html`).

Important! If you don't specify a path, most web servers will serve the file named `index.html` from that server's root folder (i.e., the path "defaults" to `index.html`). As such, this is the traditional name for the HTML file containing a websites home page.

As in any program, you should always use **relative** paths in web programming, and these paths are frequently (but not always!) relative to the web server's *root folder*.

- **`query`** (*optional*): extra **parameters** (arguments) included in the request about what resource to access. The leading `?` is part of the query.
- **`fragment`** (*optional*): indicates which part ("fragment") of the resource to access. This is used for example to let the user "jump" to the middle of a web page. The leading `#` is part of the query.

Development Servers

As noted above, it is possible to request a `.html` file (open a web page) using the `file` protocol by simply opening that file directly in the browser. This works fine for testing most client-side programs. However, there are a few client-side interactions that for security reasons only work if a web page is requested from a web server (e.g., via the `http` protocol).

For this reason, it is recommended that you develop client-side web applications using a **local development web server**. This is a web server that you run from your own computer—your machine acts as a web server, and you use the browser to have your computer send a request *to itself* for the webpage. Think about mailing yourself a letter. Development web servers can help get around cross-origin request restrictions, as well as offer additional benefits to speed development—such as *automatically reloading the web browser when the source code changes*.

There are many different ways to run a simple development server from the command-line (such as using the Python `http.server` module). These servers, when started, will "serve" files using the current directory as the "root" folder. So again, if you start a server from `/Users/joelross`, you will be able to access the `Desktop/index.html` file at `http://127.0.0.1:port/Desktop/index.html` (which port will depend on which development server you use).

- The address `127.0.0.1` is the IP address for `localhost` which is the domain of your local machine (the "local host"). Most development servers, when started, will tell you the URL for the server's root directory.
- Most commonly, you will want to start the web server from the root directory of your *project*, so that the relative path `index.html` finds the file you expect.

- You can usually stop a command-line development server with the universal `ctrl-c` cancel command. Otherwise, you'll want to leave the server running in a background terminal as long as you are working on your project.

If you use the recommended **live-server** utility, it will open a web browser to the root folder and *automatically reload the page* whenever you **save** changes to a file in that folder. This will make your life much, much better.

Chapter 3

HTML Fundamentals

A webpage on the internet is simply a set of files that the browser *renders* (shows) in a particular way, allowing you to interact with it. The most basic way to control how a browser displays content (e.g., words, images, etc) is by *encoding* that content in HTML.

HTML (**H**yper**T**ext **M**arkup **L**anguage) is a language that is used to give meaning to otherwise plain text, which the browser can then use to determine how to display that text. HTML is not a programming language but rather a *markup language*): it adds additional details to information (like notes in the margin of a book), but doesn't contain any logic. HTML is a "hypertext" markup language because it was originally intended to mark up a document with hyperlinks, or links to other documents. In modern usage, HTML describes the **semantic meaning** of content: it marks what content is the a *heading*, what content is a *paragraph*, what content is a *definition*, what content is an *image*, what content is a *hyperlink*, and so forth.

- HTML serves a similar function to Markdown, but is much more expressive and powerful.

This chapter provides an overview and explanation of HTML's syntax (how to use it to annotate content). HTML's syntax is very simple, and generally only takes someone a few days to learn—though using it effectively can require more practice.

3.1 HTML Elements

HTML content is normally written in `.html` files. By using the `.html` extension, your editor, computer, and browser should automatically know that this file will contain content marked up in HTML.

As mentioned in Chapter 2, most web servers will by default serve a file named **`index.html`**, and so that filename is traditionally used for a website's home page.

As with all programming languages, `.html` files are really just plain text files with a special extension, so can be created in any text editor. However, using a coding editor such as VS Code provides additional helpful features that can speed up your development process.

HTML files contain the **content** of your web page: the text that you want to show on the page. This content is then annotated (marked up) by surrounding it with **tags**:

Figure 3.1: Basic syntax for an HTML element.

The **opening/start tag** comes before the content and tell the computer "I'm about to give you content with some meaning", while the **closing/end tag** comes after the content to tell the computer "I'm done giving content with that meaning." For example, the `<h1>` tag represents a top-level heading (equivalent to one `#` in Markdown), and so the open tag says "here's the start of the heading" and the closing tag says "that's the end of the heading".

Tags are written with a less-than symbol **`<`**, then the name of the tag (often a single letter), then a greater-than symbol **`>`**. An *end tag* is written just like a *start tag*, but includes a forward slash `/` immediately after the less-than symbol—this indicates that the tag is closing the annotation.

- HTML tag names are not case sensitive, but you should always write them in all lowercase.
- Line breaks and white space around tags (including indentation) is ignored. Tags may thus be written on their own line, or *inline* with the content. These two uses of the `<p>` tag (which marks a *paragraph* of content) are equivalent:

```
<p>
   The itsy bitsy spider went up the water spout.
</p>
```

```
<p>The itsy bitsy spider went up the water spout.</p>
```

Taken together, the tags and the content they *contain* are called an **HTML Element**. A website is made of a bunch of these elements.

Some Example Tags

The HTML standard defines lots of different elements, each of which marks a different meaning for the content. Common elements include:

- `<h1>`: a 1st-level heading
- `<h2>`: a 2nd-level heading (and so on, down to `<h6>`)
- `<p>`: a paragraph of text
- `<a>`: an "anchor", or a hyperlink
- `<img>`: an image
- `<button>`: a button
- `<em>`: emphasized content. Note that this doesn't mean *italic* (which is not semantic), but *emphasized* (which is semantic). The same as `_text_` in Markdown.
- `<strong>`: important, strongly stated content. The same as `**text**` in Markdown
- `<ul>`: an unordered list (and `<ol>` is an ordered list)
- `<li>`: a list item (an item in a list)
- `<table>`: a data table
- `<form>`: a form for the user to fill out
- `<svg>`: a Scalable Vector Graphic (a "coded" image)
- `<circle>`: a circle (in an `<svg>` element)
- `<div>`: a division (section) of content. Also acts as an empty *block* element (followed by a line break)
- `<span>`: a span (section) of content. Also acts as an empty *inline* element (not followed by a line break)

Comments

As with every programming language, HTML includes a way to add comments to your code. It does this by using a tag with special syntax:

```
<!-- this is a comment -->
<p>this is is not a comment</p>
```

Because that syntax is somewhat awkward to type, most source-code editors will let you comment-out the currently highlighted text by pressing `cmd + /` (or `ctrl + /` on Windows). If you're using a code editor, try placing your cursor on a line and using that keyboard command to comment and un-comment the line.

Comments can appear anywhere in the file. Just as in other languages, they are ignored by any program reading the file (with a few interesting exceptions), but they do remain in the page and are visible when you view the page source.

Attributes

The start tag of an element may also contain one or more **attributes**. These are similar to attributes in object-oriented programming: they specify *properties*, options, or otherwise add additional meaning to an element. Like named parameters in R or HTTP query parameters, attributes are written in the format `attributeName=value`; values of attributes are almost always strings, and so are written in quotes. Multiple attributes are separated by spaces:

```
<tag attributeA="value" attributeB="value">
   content
</tag>
```

For example, a hyperlink anchor (`<a>`) uses a `href` ("**h**ypertext **ref**erence") attribute to specify where the content should link to:

```
<a href="https://ischool.uw.edu">iSchool homepage</a>
```

- In a hyperlink, the *content* of the tag is the displayed text, and the *attribute* specifies the link's URL. Contrast this to the same link in Markdown:

  ```
  [iSchool homepage](https://ischool.uw.edu)
  ```

Similarly, an image (`<img>`) uses the `src` (**s**ou**rc**e) attribute to specify what picture it is showing. The `alt` attribute contains alternate text to use if the browser can't show images—such as with screen readers (for the visible impaired) and search engine indexers.

```
<img src="baby_picture.jpg" alt="a cute baby">
```

- Note that because an `<img>` has no textual content, it is an *empty element* (see below).

There are also a number of global attributes that can be used on any element. For example:

- Every HTML element can include an **`id`** attribute, which is used to give them a unique identifier so that we can refer to them later (e.g., from CSS or JavaScript). `id` attributes are named like variable names, and must be **unique** on the page.

  ```
  <h1 id="title">My Web Page</h1>
  ```

  The `id` attribute is most commonly used used to create "bookmark hyperlinks", which are hyperlinks to a particular location on a page (i.e., that

cause the page to scroll down). You do this by including the `id` as the **fragment** of the URI to link to (e.g., after the `#` in the URI).

```
<a href="index.html#nav">Link to element on `index.html` with `id="nav"`</a>
<a href="#footnote">Link to element on current page with `id="footnote"`</a>
```

Note that the title attribute does NOT contain the `#` symbol, but the URI to link to does.

- The `lang` attribute is used to indicate the language in which the element's content is written. Programs reading this file might use that to properly index the content, correctly pronounce it via a screen reader, or even translate it into another language:

  ```
  <p lang="sp">No me gusta</p>
  ```

  Specify the `lang` attribute for the `<html>` element (see below) to define the default language of the page; that way you don't need to mark the language of every element. **Always include this attribute**.

  ```
  <html lang="en">
  ```

Empty Elements

A few HTML elements don't require a closing tag because they *can't* contain any content. These tags are often used for inserting media into a web page, such as with the `<img>` tag. With an `<img>` tag, you can specify the path to the image file in the `src` attribute, but the image element itself can't contain additional text or other content. Since it can't contain any content, we leave off the end tag entirely:

```
<img src="picture.png" alt="description of image for screen readers and indexers">
```

Older versions of HTML (and current related languages like XML) required you to include forward slash `/` just before the greater-than symbol. This "end" slash indicated that the element was complete and expected no further content:

```
<img src="picture.png" alt="description of image for screen readers and indexers" />
```

This is no longer required in HTML5, so feel free to omit that forward slash (though some purists, or those working with XML, will still include it).

3.2 Nesting Elements

Web pages are made up of multiple (hundreds! thousands!) of HTML elements. Moreover, HTML elements can be **nested**: that is, the content of an HTML element can contain *other* HTML tags (and thus other HTML elements):

Figure 3.2: An example of element nesting: the `<em>` element is nested in the `<h1>` element's content.

The semantic meaning indicated by an element applies to *all* its content: thus all the text in the above example is a top-level heading, and the content "(with emphasis)" is emphasized in addition.

Because elements can contain elements which can *themselves* contain elements, an HTML document ends up being structured as a **"tree"** of elements:

In an HTML document, the "root" element of the tree is always an **`<html>`** element. Inside this we put a `<body>` element to contain the document's "body" (that is, the shown content):

```
<html lang="en">
  <body>
    <h1>Hello world!</h1>
    <p>This is <em>conteeeeent</em>!</p>
  <body>
</html>
```

This model of HTML as a tree of "nodes"—along with an API (programming interface) for manipulating them— is known as the **Document Object Model (DOM)**. See Chapter: DOM for details.

Caution! HTML elements have to be "closed" correctly, or the semantic meaning may be incorrect! If you forget to close the `<h1>` tag, then *all* of the following content will be considered part of the heading! Remember to close your inner tags *before* you close the outer ones. Validating your HTML can help with this.

Block vs. Inline Elements

All HTML elements fall into one of two categories:

- **Block elements** form a visible "block" on a page—in particular, they will be on a new line from the previous content, and any content after it will also be on a new line. These tend to be structural elements for a page: headings (`<h1>`), paragraphs (`<p>`), lists (`<ul>`), etc.

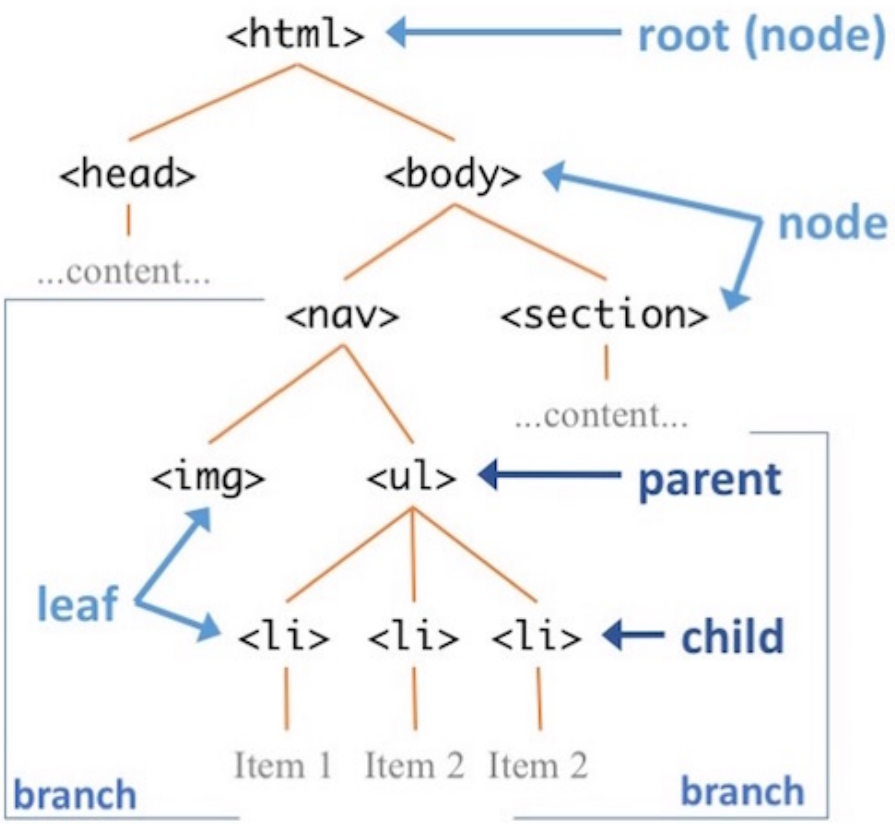

Figure 3.3: An example DOM tree (a tree of HTML elements).

```
<div>Block element</div>
<div>Block element</div>
```

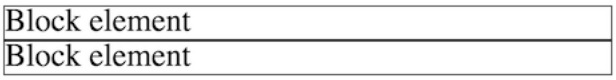

Figure 3.4: Two block elements rendered on a page.

- **Inline elements** are contained "in the line" of content. These will *not* have a line break after them. Inline elements are used to modify the content rather than set it apart, such as giving it emphasis (`<em>`) or declaring that it to be a hyperlink (`<a>`).

```
<span>Inline element</span>
<span>Other inline element</span>
```

Figure 3.5: Two inline elements rendered on a page.

Inline elements go inside of block elements, and it's common to put block elements inside of the other block elements (e.g., an `<li>` inside of a `<ul>`, or a `<p>` inside of a `<div>`). However, it is invalid to to nest a block element inside of an inline element—the content won't make sense, and probably won't look right.

Some elements have further restrictions on nesting. For example, a `<ul>` (unordered list) is *only* allowed to contain `<li>` elements—anything else is invalid markup.

3.3 Web Page Structure

Now that you understand how to specify HTML elements, you can begin making real web pages! However, there are a few more tags you need to know and include for a valid, modern web page.

Doctype Declaration

All HTML files start with a document type declaration, commonly referred to as the "Doctype." This tells the rendering program (e.g., the browser) what format and syntax your document is using. Since you're writing pages with HTML 5, you can declare it as follows:

```
<!DOCTYPE html>
<html lang="en">
    ...
</html>
```

`<!DOCTYPE>` isn't technically an HTML tag (it's actually XML). While modern browsers will perform a "best guess" as to the Doctype, it is best practice to specify it explicitly. Always include the DOCTYPE at the start of your HTML files!

The `<head>` Section

In addition to the `<body>` element that defines the displayed content, you should also include a `<head>` element that acts as the document "header" (the `<head>` is nested inside the `<html>` at the same level as the `<body>`). The content of the `<head>` element is *not* shown on the web page—instead it provides extra (meta) information *about* the document being rendered.

There are a couple of common elements you should include in the `<head>`:

- A **`<title>`**, which specifies the "title" of the webpage:

  ```
  <title>My Page Title</title>
  ```

  Browsers will show the page title in the tab at the top of the browser window, and use that as the default bookmark name if you bookmark the page. But the title is *also* used by search indexers and screen readers for the blind, since it often provides a strong signal about what the page's subject. Thus your title should be informative and reflective of the content.

- A `<meta>` tag that specifies the character encoding of the page:

  ```
  <meta charset="UTF-8">
  ```

  The `<meta>` tag itself represents "metadata" (information about the page's data), and uses an attribute and value to specify that information. The most important `<meta>` tag is for the character set, which tells the browser how to convert binary bits from the server into letters. Nearly all editors these days will save files in the `UTF-8` character set, which supports the mixing of different scripts (Latin, Cyrillic, Chinese, Arabic, etc) in the same file.

- You can also use the `<meta>` tag to include more information about the author, description, and keywords for your page:

  ```
  <meta name="author" content="your name">
  <meta name="description" content="description of your page">
  <meta name="keywords" content="list,of,keywords,separate,by,commas">
  ```

Note that the `name` attribute is used to specify the "variable name" for that piece of metadata, while the `content` attribute is used to specify the "value" of that metadata. `<meta>` elements are *empty elements* and have no content of their own.

Again, these are not visible in the browser window (because they are in the `<head>`!), but will be used by search engines to index your page.

  - *At the very least, always include author information for the pages you create!*

- We will discuss additional elements for the `<head>` section throughout the text, such as using `<link>` to include CSS and using `<script>` to include JavaScript.

3.4 Web Page Template

Putting this all together produces the following "template" for making a web page:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="author" content="your name">
   <meta name="description" content="description of your page">
   <title>My Page Title</title>
</head>
<body>
   ...
   Content goes here!
   ...
</body>
</html>
```

You can use this to start off every web page you ever create from now on!

Resources

- Getting Starting with HTML
- General HTML 5 Reference
- Alphabetical HTML Tag Reference
- Dive into HTML5 - Free book on HTML
- W3C HTML Validation Service

Also remember you can view the HTML page source of *any* webpage you visit. Use that to explore how others have developed pages and to learn new tricks and techniques!

Chapter 4

CSS Fundamentals

CSS (**C**ascading **S**tyle **S**heets) is a declarative language used to alter the appearance or *styling* of a web page. CSS is used to define a set of formatting **rules**, which the browser applies when it renders your page. Thus CSS can tell the browser to use a particular *font* for the page text, a certain *color* for the first paragraph in an article, or a picture for the page's *background*.

Files of CSS rules (called **stylesheets**) thus act kind of like Styles or Themes in PowerPoint, but are way more powerful. You can control nearly every aspect of an element's appearance, including its overall placement on the page.

- To give you some idea of just how much control you have, check out the examples in the CSS Zen Garden. Every one of those examples uses the exact same HTML page, but they all look completely different because each one uses a different CSS stylesheet.

This chapter will explain how to include CSS in your web page and the overall syntax for declaring basic CSS rules. Additional details and options can be found in the following chapter

4.1 Why Two Different Languages?

If you are new to web programming, you might be wondering why there are two different languages: HTML for your page content; and CSS for formatting rules. Why not just include the formatting right in with the content?

There is an old, tried-and-true principle in programming referred to as **"separation of concerns"**. Well-designed software keeps separate things separate, so that it's easy to change one without breaking the other. And one of the most common forms of separation is to keep the **data** (content) in a program separate from the **presentation** (appearance) of that data.

By separating content (the HTML) from its appearance (the CSS), you get a number of benefits:

- The same content can easily be presented in different ways (like in the CSS Zen Garden). In web development, you could allow the user to choose different "themes" for a site, or you could change the formatting for different audiences (e.g., larger text for vision-impaired users, more compact text for mobile users, or different styles for cultures with different aesthetic sensibilities).
- You can have several HTML pages to all share the same CSS stylesheet, allowing you to change the look of an entire web site by only editing one file. This is an application of the Don't Repeat Yourself (DRY) principle.
- You can also dynamically adjust the look of your page by applying new style rules to elements in response to user interaction (clicking, hovering, scrolling, etc.)
- Users who don't care about about the visual appearance (e.g., blind users with screen readers, automated web indexers) can more quickly and effectively engage with the content without needing to determine what information is "content" and what is just "aesthetics".

Good programming style in web development thus keeps the **semantics** (HTML) separate from the **appearance** (CSS). Your HTML should simply describe the meaning of the content, not what it looks like!

For example, while browsers might normally show `<em>` text as italic, we can use CSS to instead make emphasized text underlined, highlighted, larger, flashing, or with some other appearance. The `<em>` says nothing about the visual appearance, just that the text is emphatic, and it's up to the styling to determine how that emphasis should be conveyed *visually*.

4.2 CSS Rules

While it's possible to write CSS rules directly into HTML, the best practice is to create a separate CSS **stylesheet** file and connect that to your HTML content. These files are named with the `.css` extension, and are typically put in a `css/` folder in web page's project directory, as with the following folder structure:

```
my-project/
|-- css/
   |-- style.css
|-- index.html
```

(`style.css`, `main.css`, and `index.css` are all common names for the "main" stylesheet).

You connect the stylesheet to your HTML by adding a **`<link>`** element to your page's `<head>` element:

```
<head>
  <!--... other elements here...-->

  <link rel="stylesheet" href="css/style.css">
</head>
```

The `<link>` element represents a connection to another resource. The tag includes an attribute indicating the **rel**ation between the resources (e.g., that the linked file is a stylesheet). The `href` attribute should be a *relative path* from the `.html` file to the `.css` resource. Note also that a `<link>` is an empty element so has no closing tag.

Overall Syntax

A CSS stylesheet lists **rules** for formatting particular elements in an HTML page. The basic syntax looks like:

```
/* this is pseudocode for a CSS rule */
selector {
   property: value;
   property: value;
}

/* this would be another, second rule */
selector {
   property: value;
}
```

A CSS **rule** rule starts with a **selector**, which specifies which elements the rule applies to. The selector is followed by a pair of braces **`{}`**, inside of which is a set of formatting **properties**. Properties are made up of the property *name* (e.g., `color`), followed by a colon (**`:`**), followed by a *value* to be assigned to that property (e.g., `purple`). Each name-value pair must end with a semi-colon (**`;`**).

- If you forget the semi-colon, the browser will likely ignore the property and any subsequent properties—and it does so silently without showing an error in the developer tools!

Like most programming languages, CSS ignores new lines and whitespace. However, most developers will use the styling shown above, with the brace on the same line as the selector and indented properties.

As a concrete example, the below rule applies to any `h1` elements, and makes them appear in the 'Helvetica' font as white text on a dark gray background:

```
h1 {
  font-family: 'Helvetica';
  color: white;
  background-color: #333; /*dark gray*/
}
```

Note that CSS **comments** are written using the same block-comment syntax used in Java (`/* a comment */`), but *cannot* be written using inline-comment syntax (`//a comment`).

When you modify a CSS file, you will need to *reload the page in your browser* to see the changed appearance. If you are using a program such as `live-server`, this reloading should happen automatically!

CSS Properties

There are many, many different CSS formatting properties you can use to style HTML elements. All properties are specified using the `name:value` syntax described above—the key is to determine the name of the property that produces the appearance you want, and then provide a valid value for that property.

Pro-tip: modern editors such as VS Code will provide auto-complete suggestions for valid property names and values. Look carefully at those options to discover more!

Below is a short list of common styling properties you may change with CSS; more complex properties and their usage is described in the following chapter.

- `font-family`: the "font" of the text (e.g., `'Comic Sans'`). Font names containing white space *must* be put in quotes (single or double), and I tend to quote any specific font name as well.

  Note that the value for the `font-family` property can also be a *comma-separated list* of fonts, with the browser picking the first item that is available on that computer:

  ```
  /* pick Helvetic Nue if exists, else Helvetica, else Arial, else the default
     sans-serif font */
  font-family: 'Helvetica Nue', 'Helvetica', 'Arial', sans-serif;
  ```

- `font-size`: the size of the text (e.g., `12px` to be 12 pixels tall). The value must include units (so `12px`, not `12`). See the next chapter for details on Units & Sizes.

- `font-weight`: boldness (e.g., `bold`, or a numerical value such as `700`).

- `color`: text color (e.g., either a named color like `red` or a hex value like `#4b2e83`. See the next chapter for details on colors. The

`background-color` property specifies the background color for the element.

- `border`: a border for the element (see "Box Model" in the next chapter). Note that this is a short-hand property which actually sets multiple related properties at once. The value is thus an *ordered* list of values separated by **spaces**:

```
/* border-width should be 3px, border-style should be dashed, and border-color
   should be red */
border: 3px dashed red;
```

Read the documentation for an individual property to determine what options are available!

Note that not all properties or values be effectively or correctly supported by all browsers. Be sure and check the browser compatibility listings!

CSS Selectors

Selectors are used to "select" which HTML elements the css rule should apply to. As with properties, there are many different kinds of selectors (and see the following chapter), but there are three that are most common:

Element Selector

The most basic selector, the **element selector** selects elements by their element (tag) name. For example, the below rule will apply the all `<p>` elements, regardless of where they appear on the page:

```
p {
   color: purple;
}
```

You can also use this to apply formatting rules to the entire page by selecting the `<body>` element. Note that for clarity/speed purposes, we generally do *not* apply formatting to the `<html>` element.

```
body {
   background-color: black;
   color: white;
}
```

Class Selector

Sometimes you want a rule to apply to only *some* elements of a particular type. You will most often do this by using a **class selector**. This rule will select elements with a `class` attribute that contains the specified name. For example, if you had HTML:

```
<!-- HTML -->
<p class="highlighted">This text is highlighted!</p>
<p>This text is not highlighted</p>
```

You could color just the correct paragraph by using the class selector:

```
/* CSS */
.highlighted {
   background-color: yellow;
}
```

Class selectors are written with a single dot (`.`) preceding the *name of the class* (not the name of the tag!) The `.` is only used in the CSS rule, not in the HTML `class` attribute.

Class selectors also let us apply a single, consistent styling to multiple different types of elements:

```
<!-- HTML -->
<h1 class="alert-flashing">I am a flashing alert!</h1>
<p class="alert-flashing">So am I!<p>
```

CSS class names should start with a letter, and can contain hyphens, underscores, and numbers. Words are usually written in lowercase and separated by hyphens rather than camelCased.

Note that HTML elements can contain **multiple classes**; each class name is separate by a **space**:

```
<p class="alert flashing">I have TWO classes: "alert" and "flashing"</p>
<p class="alert-flashing">I have ONE class: "alert-flashing"</p>
```

The class selector will select any element that *contains* that class in its list. So the first paragraph in the above example would be selected by either `.alert` **OR** `.flashing`.

You should always strive to give CSS classes **semantic names** that describe the purpose of element, rather than just what it looks like. `highlighted` is a better class name than just `yellow`, because it tells you what you're styling (and will remain sensible even if you change the styling later). Overall, seek to make your class names *informative*, so that your code is easy to understand and modify later.

There are also more formal methodologies for naming classes that you may wish to utilize, the most popular of which is BEM (Block, Element, Modifier).

Class selectors are often commonly used with `<div>` (block) and `<span>` (inline) elements. These HTML elements have *no* semantic meaning on their own, but can be given appearance meaning through their `class` attribute. This allows them to "group" content together for styling:

```
<div class="cow">
  <p>Moo moo moo.</p>
  <p>Mooooooooooooooooooo.</p>
</div>

<div class="sheep">
  <p>Baa baa <span class="dark">black</span> sheep, have you any wool?</p>
</div>
```

Id Selector

It is also possible to select HTML elements by their `id` attribute by using an **id selector**. Every HTML element can have an `id` attribute, but unlike the `class` attribute the value of the `id` must be unique within the page. That is, no two elements can have the same value for their `id` attributes.

Id selectors start with a `#` sign, followed by the value of the `id`:

```
<div id="sidebar">
   This div contains the sidebar for the page
</div>
```

```
/* the one element with id="sidebar" */
#sidebar {
    background-color: lightgray;
}
```

The `id` attribute is more specific (it's always just one element!) but less flexible than the `class` attribute, and makes it harder to "reuse" your styling across multiple elements or multiple pages. Thus you should *almost always use a class selector instead*, unless you are referring to a single, specific element.

4.3 The Cascade

CSS is called **Cascading** Style Sheets because multiple rules can apply to the same element (in a "cascade" of style!)

CSS rules are *additive*—if multiple rules apply to the same element, the browser will combine all of the style properties when rendering the content:

```
/* CSS */
p { /* applies to all paragraphs */
  font-family: 'Helvetica'
}

.alert { /* applies to all elements with class="alert" */
  font-size: larger;
}

.success { /* applies to all elements with class="success" */
  color: #28a745; /* a pleasant green */
}
```

```
<!-- HTML -->
<p class="alert success">
  This paragraph will be in Helvetica font, a larger font-size, and green color,
  because all 3 of the above rules apply to it.
</p>
```

CSS styling apply to *all* of the content in an element. And since that content can contain other elements that may have their own style rules, rules may also in effect be *inherited*:

```
<div class="content"> <!-- has own styling -->
  <div class="sub-sec"> <!-- has own styling + .content styling -->
    <ol class="demo-list"> <!-- own styling (ol AND .demo-list rules) + .sub-sec + .

      <!-- li styling + .demo-list + .sub-sec + .content -->
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ol>
  </div>
</div>
```

We call these inherited properties, because the child elements inherit the setting from their ancestor elements. This is a powerful mechanism that allows you to specify properties only once for a given branch of the DOM element tree. In general, try to set these properties on the highest-level element you can, and let the child elements inherit the setting from their ancestor.

Rule Specificity

Important! Rules are applied in the order they are defined in the CSS file. If you link multiple CSS files to the same page, the files are processed in order they are linked in the HTML. The browser selects elements that match the rule and applies the rule's property. If a later rule selects the same element and applies a different value to that property, the previous value is *overridden*. So in general, all things being equal, **the last rule on the page wins**.

```
/* two rules, both alike in specificity */
p { color: red; }
p { color: blue; }
```

```
<p>This text will be blue, because that rule comes last!</p>
```

However, there are some exceptions when CSS treats rules as *not* equal and favors earlier rules over later ones. This is called Selector Specificity. In general, more specific selectors (`#id`) take precedence over less specific ones (`.class`, which is more specific than `tag`). If you notice that one of your style rules is not being applied, despite your syntax being correct, check your browser's developer tools to see if your rule is being overridden by a more specific rule in an earlier stylesheet. Then adjust your selector so that it has the same or greater specificity.

```
.alert { color: red; }
div { color: blue; }
```

```
<div class="alert">This text will be red, even though the `div` selector is last,
because the `.alert` selector has higher specificity so is not overridden.</div>
```

Precedence rules are **not** a reason to prefer `#id` selectors over `.class` selectors! Instead, you can utilize the more complex selectors described in Chapter 6 to be able to create reusable rules an avoid duplicating property declarations.

Resources

- Getting started with CSS (MDN)
- CSS Tutorial (w3schools)
- CSS Reference (MDN) a complete alphabetical reference for all CSS concepts.
- CSS Selectors Reference a handy table of CSS selectors.
- CSS Properties Reference a table of CSS properties, organized by category.
- CSS-Tricks a blog about tips for using CSS in all kinds of ways. Contains many different useful guides and explanations.
- W3C CSS Validation Service

Chapter 5

Standards and Accessibility

This chapter details considerations and techniques to ensure that your web pages and applications can be used by ***everyone***, regardless of their choice of web browser or their own physical abilities.

5.1 Web Standards

As discussed in Chapter 2, the HTML and CSS files you author are delivered to clients upon request. The code within these files is *interpreted* by the **web browser** in order to create the visual presentation that the user can see and interact with.

A web browser is thus any piece of software capable of rendering these `.html` and `.css` files (and sending HTTP requests to fetch them in the first place). And there are many different web browsers in the world:

- Note that while there may be some clear "winners" in terms of browser popularity, do not dismiss less popular browsers. For example, if 0.34% of users use Internet Explorer 8 (July 2017), that's roughly 1,300,000 people worldwide.

These web browsers are all created by different developers, working for different (often rival!) organizations. How is it that they are all able to read and interpret the same code, and produce the same rendered output? *Standards.*

Web Standards are agreed-upon specifications for how web page source code should be rendered by the browser. Web standards detail both the language syntax (e.g., how to write HTML tags) and the language semantics (e.g., which HTML tags to use), so that it can be understood by any browser that follows (agrees to) that standard. And since as a developer you want your pages to

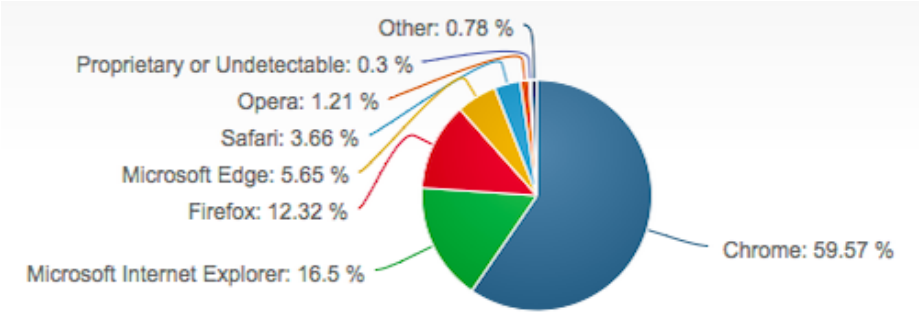

Figure 5.1: *Desktop* browser market share July 2017, from netmarketshare.com. See also caniuse.com's usage table for information on mobile and older browsers.

render the same across all browsers, web standards give the requirements for how you should write your code so that your pages render correctly.

Modern web standards are created and maintained by a huge group of stakeholders known as the World Wide Web Consortium (W3C), which includes major browser developers such as Google and Mozilla. However, this group has no enforcement powers: and so browsers often deviate from the published standards! A browser may ignore a standard to "help out" developers (e.g., making a "best guess" to render malformed HTML content), or to introduce new features (e.g., a new CSS property that produces some special effect). Some browsers are better at conforming to the accepted standards than others. Internet Explorer—IE 6 in particular—is notorious for not meeting standards and requiring extra effort from developers to make pages work. This is part of what IE has such a bad reputation and gets so much scorn from developers (though Microsoft Edge, the browser in Windows 10, is awesome about being standards compliant).

Getting so many people to agree on a standard of communication takes time; thus web standards change relatively slowly: the HTML 4 standard was adopted in 1997, but the **HTML 5** standard (that this course teaches) wasn't finalized until 2014. The **CSS 3** standard is broken up into more than 50 different *modules* that are developed and introduced independently, and so is continuously being adopted piece-wise.

When introducing new or experimental CSS properties, browsers historically utilized vendor prefixes in naming the properties. As standards are being decided upon, each browser may treat that property in a slightly different way, thus forcing developers who want to use the "latest and greatest" to provide a different definition or value for each vendor. Prefixes are a naming convention that will cause the property to only be understood and applied by a particular browser; e.g., `-webkit-hyphens` would use the Webkit version of `hyphens` property (for controlling word breaks), while `-ms-hyphens` would use the IE version. This

practice is currently discouraged and being fazed out, though prefixes may be required when supporting older browsers. Tools such as Autoprefixer can help automatically manage prefixes.

In general, as long as you program webpages that conform to web standards, they will work on all "*modern browsers*"—though there may still be a few cross-browser compatibility concerns to notice. You should thus test and **validate** your code against the standards. Luckily, there the W3C provides online tools that can help validate code:

- W3C HTML Validation Service
- W3C CSS Validation Service
- W3C Developer Tools for a complete list of validators

To use these services, simply enter your web pages URL (or copy and paste the contents of your `.html` or `.css` files), and then run the validation. You will definitely want to fix any *errors* you get. *Warnings* should be considered; however, it is possible to get false positives. Be sure and read the warning carefully and consider whether or not it is actually a "bug" in your code!

Supporting *really* old browsers that are not standards compliant is left as an exercise to the reader.

Develop for other people's browsers, not your own! Just because it "works" for you, doesn't mean it works for anyone else. Test your code against standards and automated systems; don't just look at the rendered result in a single browser!

5.2 Why Accessibility

Consider the following hypothetical webpage user:

> Tracy is a 19-year-old college student and was born blind. In high school she did well as she could rely on audio tapes and books and the support of tutors, so she never bothered to really learn Braille. She is interested in English literature and is very fond of short stories; her dream is to become an audiobook author. Tracy uses the Internet to share her writing and to connect with other writers through social networks. She owns a laptops and uses a screen reader called JAWS: a computer program which reads her screen out loud to her in an artificial voice. (Adapted from here)

One of the most commonly overlooked limitations on a website's usability is whether or not it can be used by people with some form of disability. There are many different forms of disabilities or impairments that may affect whether or not a person can access a web page, including:

- *Vision Impairments*: About 2% of the population is blind, so use alternate mediums for reading web pages. Farsightedness and other vision problems are also very common (particularly among older adults), requiring larger and clearer text. Additionally, about 4.5% the population is color-blind.
- *Motor Impairments*: Arthritis occurs in about 1% of the population, and can restrict people's ease at using a mouse, keyboard or touch screen. Other impairments such as tremors, motor neuron conditions, and paralysis may further impact people's access.
- *Cognitive Impairments*: Autism, dyslexia, and language barriers may cause people to be excluded from utilizing your website.

If you fail to make your website accessible, you are locking out 2% or more of users, reducing the availability and use of your website. Indeed, companies are finally seeing this population as an important but excluded market; for example, Facebook has an entire team devoted to accessibility and supporting users with disabilities. "*Accessibility Engineers*" have good job prospects.

Supporting users with disabilities is not just the morally correct thing to do, it's also *the law*. US Courts have ruled that public websites are subject to Title III of the Americans with Disabilities Act (ADA), meaning that is a possible and not uncommon occurrence for large organizations to be sued for discrimination if their websites are not accessible. So far, "accessibility" has legally meant complying with the W3C's Web Content Accessibility Guidelines (WCAG) (see below), a standard that is not overly arduous to follow if you consider accessibility from the get-go.

Finally, designing your website (or any system) with accessibility in mind will not just make it more usable for those with disabilities—it will make it more usable for *everyone*. This is the principle design **Universal Design** (a.k.a. *universal usability*): designing for *accessibility*—being usable by all people no matter their ability (physical or otherwise)—benefits not just those with some form of limitation or disability, but **everyone**. The classic real-world example of universal design are curb cuts: the "slopes" built into curbs to accommodate those in wheelchairs. However, these cuts end up making life better for *everyone*: people with rollerbags, strollers, temporary injuries, small children learning to ride a bicycle, etc.

Universal design applies to websites as well:

- If you support people who can't see, then you may also support people who can't see *right now* (e.g., because of a bad glare on their screen).
- If you support people with motor impairments, then you may also support people trying to use your website without a mouse (e.g., from a laptop while on a bumpy bus).
- If you support people with cognitive impairments, then you may also support people who are temporarily impaired (e.g., inebriated or lacking sleep).

If you make sure that your web page is well-structured and navigable by those

with screen readers, it will ensure that it is navigable by *other* machines, such as search engine indexers. Or for unusual or future browsers (a virtual reality browser perhaps).

Thus supporting accessibility in client-side web development is important both for helping a population that is often overlooked (a form of social justice), as well as for supporting new technologies and systems.

In addition to individual capabilities, people are also reliant on a large amount of existing *infrastructure* to ensure that they have an internet connection and their requests can reach your web server. The lack of such access is often tied to economic or social inequalities, forming what is called the *digital divide*. Considering the availability of network access and other infrastructural needs is vitally important when developing information technologies, but is difficult to manage through client-side development (though see Responsive Design for some things you can do).

5.3 Supporting Accessibility

In this course, we will primarily aim to support accessibility for users with visual impairments such as those using screen readers. A **screen reader** is a piece of software that is able to synthesize and "read" content on a computer's screen out loud through the speakers, so that users are able to navigate and control the computer without needing to see the screen. Screen readers are often combined with *keyboard controls*, so that users use just the keyboard to control the computer and not the mouse (almost like a command-line interface!).

There are a number of different screen reader software packages available:

- Macs have had VoiceOver built into the operating system since 2005, though it has been refined with each new OS version.
- On Windows, the most popular screen readers are JAWS and NDVA. Windows also has a built-in screen reader called Microsoft Narrator, which with Windows 10 is beginning to reach parity with the 3rd-party offerings.

You should try out this software! Follow the above links to learn how to turn on the screen reader for your computer, and then try using it to browse the internet *without looking at the screen*. This will give you a feel for what it is like using a computer while blind.

Screen readers are just software that interpret the HTML of a website in order to allow the user to hear and navigate the content—they are basically *non-visual web browsers*. As such, supporting screen readers just means implementing your web site so it works with this browser. In particular, this means making sure your page conforms to the Web Accessibility Content Guidelines (WCAG). This is a list of principles and techniques that you should utilize when authoring

HTML document in order to make sure that they are accessible. The guidelines are driven by 4 main principles:

1. **Perceivable**: Information and user interface components must be presentable to users in ways they can perceive.
2. **Operable**: User interface components and navigation must be operable.
3. **Understandable**: Information and the operation of user interface must be understandable.
4. **Robust**: Content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technologies.

More concretely, accessible web pages are those that can be **navigated** by screen readers (so people can easily get to the information they care about), and have content can be **perceived** by screen readers (so is not just presented visually).

Don't worry; there are simple, specific implementation steps you can to follow these principles, the most common of which are described below.

Semantic HTML

A major step in making sure your web pages are accessible is to make sure that your use of HTML elements is *semantic*. HTML elements should be used to describe the meaning or form of their content, *not* to give that content a visual appearance!

For example, an `<h1>` element is used to indicate a top-level heading, such as the title of the page. But by default, browsers will give `<h1>` elements a different visual appearance than unmarked content (usually by making it larger and bold). It is possible to achieve a similar visual effect just using CSS:

```
<!-- html -->
<h1>This is a real heading!</h1>
<div class="fake-header">
    This just LOOKS like a heading.
</div>
```

```
/* css */
.fake-header {
  font-size: 2em;
  font-weight: bold;
}
```

This is a real heading!

This just LOOKS like a heading.

Figure 5.2: A real heading and a fake heading

Your HTML should **always** be semantic! A screen reader will (mostly) ignore the CSS, so if something is supposed to be a top-level heading, you need to tag it as such for the screen reader to understand that. Similarly, you should **only** use elements when they are semantically appropriate—don't use an `<h3>` element just because you want text to look bigger if it isn't actually a third-level heading (use CSS to do the styling instead).

HTML is just for semantics. CSS is for appearance.

This is the same reason that you should never use the `<i>` tag to make elements *italic*. The `<i>` tag has no valuable semantic meaning (it just means "italic", or "different"); instead you want to use a more meaningful semantic tag. Usually this means `<em>`, since italic text is usually being *emphasized*.

In fact, HTML 5 includes a large number of semantic elements that can be used to indicate "formatted" text that has a particular meaning, including ones such as: `<abbr>`, `<address>`, `<cite>`, `<del>`, `<dfn>`, `<mark>`, and `<time>`. Be sure and check the documentation for each of these elements to determine when they are supposed to be used (as well as what additional attributes they may support).

Never use the `<table>` tag to structure documents and position content. The `<table>` tag should **only** be used for content that is semantically a table (e.g., a data table). If you want to lay out content in a grid, use a CSS system such as Flexbox (see Chapter 7).

ARIA

The WCAG is developed by the Web Accessibility Initiative (WAI), who *also)* have authored an additional **web standard** for supporting accessibility. This standard is called the Accessible Rich Internet Applications Suite, or **ARIA**. ARIA specifies an *extension* to HTML, defining additional HTML *attributes* that can be included in elements in order to provide additional support for screen readers. For example, ARIA provides additional support to help screen readers navigate through a page (for people who can't just visually scroll), as well as providing a mechanism by which "rich" interactive web apps (such as those you will create with JavaScript) can communicate their behavior to screen readers.

In short, where the HTML standard falls short in supporting accessibility, ARIA steps in.

For example, the ARIA specification defines a **`role`** attribute that allows non-semantic elements (such as `<div>` elements) to specify their semantic purpose. For example:

```
<div class="btn" role="button">Submit</div>
```

would define a unsemantic `div` element that is *styled* (via a developer-defined CSS class) to look like a button. But in order to tell the screen reader that this element can be clicked like a button, it is given the `role` of `"button"`. There are a large number of different roles (following a potentially complex classification scheme), and include roles for interactive widgets such as a popup `dialog`, a `progressbar`, or a `menu`. These are important when making complex interactions, though for most simpler pages you can stick with the standard semantic elements (e.g., a `<button>` instead of a custom `<div>`).

The other important type of ARIA `role` are landmark roles. These are roles that are used to indicate specific locations in the document, to allow the screen reader to easily "jump" to different sections of the content (thus making the page *navigable*). Landmark roles include:

- `banner`: the "banner" at the top of the page—that is, content related to the website rather than to the specific page itself. Alternatively, use the `<header>` element described below (in which case you don't need to include the `role="banner"` attribute, as it is implied).
- `navigation`: a section of links for moving around the page, such as a navigation bar. This allows screen readers to quickly jump to the "quick links" and move around the page as quickly as possible. Note though that listening to all of the links read can be time consuming, so allowing the user to jump to the rest of the content is also important. Alternatively, use the `<nav>` element described below.
- `main`: the start of the "main" content (usually after the banner). This is useful for allowing the screen reader to jump to the meat of the page, past the banner and navigation sections. Alternatively, use the `<main>` element described below.
- `contentinfo`: a section that contains information about the webpage, such as author contact info and copyright information. This is usually found at the bottom of the page. Alternatively, use the `<footer>` element described below.

Considering your content in terms of these roles is a good design trick for figuring out how to organize your page.

Page Structure (Navigable)

Because screen readers cannot take in a web page's content "at a glance", accessible pages need to be explicitly structured so that they their content can be easily navigated—e.g., so that the user can quickly "scroll" to a particular blog post.

The most important way to provide this structure is by the considered use of **heading** elements (`<h1>`, `<h2>`, etc). Screen readers automatically generate a "table of contents" based on these headings, allowing users to easily move through large amounts of content. In order to be sure that the headings are

useful, they need to be **meaningful** (actually marking section headings) and **hierarchical** (they don't skip levels: every `<h3>` has an `<h2>` above it). The former is just good HTML usage that you will be doing anyway; the later may take some consideration.

The `<h#>` heading elements are part of the original HTML specification, and so will be supported by *all* screen reader systems. However, HTML 5 introduced additional elements that can be used to help organize web page content in order to make its structure more explicit. These are often referred to as semantic elements or "sectioning elements". These are all **block-level** elements that produce no visual effects on their own, but provide semantic structuring to web content. You can think of them as specialized `<div>` elements.

One metaphor is that nested HTML elements into a `<div>` is like putting that content into a envelope to "group" it together (e.g., for styling purposes). In that case, a semantic element such as `<header>` is just an envelope with a unique color that makes it easier to find in the filing cabinet.

- `<header>` represents the "header" or introduction part of the page, such as the title or banner image. It may also include common page elements such as navigation or search bars. This corresponds to the `role="banner"` landmark role; you do not need to include that role if you use this element.

  Note that a `<header>` is different from a head**ing** (`<h1>`) which is different from the `<head>`! An *head**ing*** is element (e.g., `<h1>`) that includes a title or subtitle. The *<head**er***> is a "grouping" element that can contain multiple elements, and usually has banner/logo information. The `<head>` is an element that is NOT part of the `<body>` (so is not shown in the web page), and contains *metadata* about that page.

- `<nav>` contains navigation links, usually for navigating around the site (think like a navigation bar). Not all links need to be in a `<nav>`; this is for "sections" of the webpage that are purely navigational. This element corresponds to the `role="navigation"` landmark role.

- `<main>` represents the "main" content of the document. This usually comes *after* the `<header>` (but not inside—in fact, `<main>` *cannot* be a descendant of `<header>`). Note that a web page can only have a single `<main>` element. This element corresponds to the `role="main"` landmark role.

- `<footer>` represents the "footer" of the page, usually containing information about the page. This element corresponds to the `role="contentinfo"` landmark role.

- `<section>` represents a standalone section of content (e.g., that might have a subheading such as `<h3>`). A `<section>` can also contain its own `<header>`, `<footer>`, and `<nav>` elements relevant to that section.

  Similarly, an `<article>` element also represents standalone content, but

content that might be published independently (such as a news article or a blog post). Note that a `<section>` may group together multiple `<article>` elements (such as a blog roll), and an `<article>` might potentially contain more than one `<section>`. Think about how a newspaper is structured (with a "Sports Section" that contains articles, which may themselves have different sections).

Overall, utilizing these semantic sectioning elements will help organize your content for screen readers (so visually impaired users can easily navigate the page), as well as making your content more clearly structured.

Visual Information (Perceivable)

Webpages often contain a significant amount of *primarily visual* information. In addition to obvious media such as images or video, interactive elements may be labeled visually (e.g., a search button that shows a magnifying glass). In order to make web pages accessible, screen readers need a way to **perceive** and correctly interpret such content.

The most common form of visual information are images (created with the `<img>` element). In order to make images accessible, you should always include an `alt` attribute that gives **alt**ernate text for when the images cannot be displayed (e.g., on screen readers, but also if the image fails to load):

```
<img src="baby_picture.jpg" alt="a cute baby">
```

This will be read by screen readers as "a cute baby, image". Note that the "alt-text" should not include introductory text such as *"a picture of"*, as screen readers will already report that something is an image!

Every image should include an `alt` attribute!

For more complex images (such as charts or infographics), you can instead provide a *link* to a longer description by using the `longdesc` attribute. This attribute takes a value that is a URI (relative or absolute; what you would put in the `href` attribute of a hyperlink) referring to where the description can be found. Screen readers will prompt the user with the option to then navigate to this long description.

```
<img src="accessibility_infographic.png"
     alt="an infographic showing how to make web pages accessible"
     longdesc="infographic_text.html"> <!-- link to other page with text description
```

Of course, including descriptive text is good for *any* user, not just the visually impaired! If you wish to add a caption to an image, you can do so accessibly by placing the image inside a `<figure>` element, and then using a `<figcaption>` element to semantically designate the caption.

```
<figure>
    <img src="chart.png" alt="a chart showing some information">
    <figcaption>
        A caption for the above figure. It provides the same information,
        but in a text format.
    </figcaption>
</figure>
```

(The `<figure>` element is another good example of how multiple HTML elements may be nested and combined to produce complex but accessible page structure!)

On the other hand, some images (or other elements) are purely decorative: company logos, icons that accompany text descriptions on buttons, etc. You can cause a screen reader to "skip" these elements by using an ARIA attribute `aria-hidden`. The element will still appear on the page, it just will be ignored by screen readers.

```
<!-- a search button with an icon -->
<!-- the icon will not be read, but the button text will be -->
<button><img src="search_icon.png" aria-hidden="true">Search</button>
```

It is also possible to use ARIA attribute to provide an equivalent to an `alt` tag for elements other than `<img>`, such as a `<div>` that may be styled in a purely visual way (perhaps with a background image). The **`aria-label`** acts like the alt attribute for any element, specifying what text should be read *in place of the normal content*.

```
<div class="green-rect" aria-label="a giant green rectangle"></div>
```

For longer descriptions, use the **`aria-describedby`** attribute to include a reference to the `id` of a different element *on the same page* that contains the textual description. `aria-describedby` takes as a value a fragment reference (similar to a bookmark hyperlink) to the element containing the description.

```
<div class="green-rect" aria-describedby="#rectDetail"></div>
<p id="rectDetail">The above rectangle is giant and green.</p>
```

As with bookmark hyperlinks, notice that the `aria-describedby` value starts with a `#`, but the target element's `id` does not.

Additionally, note that while `aria-label` attributes *replaces* its elements content, the `aria-describedby` attribute will be read *after* the element's content (see also here).

ARIA provides a number of other attributes that can be used to control screen readers, but these are the most common and useful.

Overall, the way to make purely visual information accessible is to *not have purely visual information*. Always include textual descriptions and captions for

images, prefer text labels for buttons, etc.

In conclusion, creating accessible content basically means writing proper and semantic HTML (what you would do doing anyway), with the small extra step of making sure that visual content is also labeled and perceivable. By meeting these standards, you will ensure that your website is usable by everyone.

Resources

- Caniuse.com details about browser support for emerging web features
- W3C Validation Tools
- Website Accessibility and the Law summative blog post, May 2017
- **Teach Access Tutorial** for creating accessible web pages. Provides lost of examples and details. Be sure and check out the code checklist and the design checklist.
- UW Accessibility Checklist a thorough and detailed list of considerations for developing accessible systems put together by UW. See also their complete list of tools and resources.
- ARIA Resources (MDN)
- WAVE Accessibility Evaluation Service

Chapter 6

More CSS

Chapter 4 explained the basic syntax and usage of CSS, enough to let you create and style your own web pages. This chapter provides more details about additional selectors and properties to use when defining CSS rules; the following chapter discusses particular properties that can be used to further style the layout of your page's content.

6.1 Compound Selectors

As described in the previous chapter, the core selectors used in CSS are the **element selector**, **class selector**, and **id selector**. However, CSS does offer ways to combine these selectors in order to specify rules only for particular elements or groups of elements.

Group Selector

The **group selector** allows you to have a single rule apply to elements matched by lots of different selectors. To do this, separate each selector with a comma (**,**); the properties defined in the rule will then apply to any element that is matched by *any* of the selectors. For example, if you want to have a single style for all headings, you might use:

```
/* applies to h1, h2, AND h3 tags */
h1, h2, h3 {
    font-family: Helvetica;
    color: #4b2e83; /* UW purple */
}
```

The comma-separated selectors can by **any** kind of selector, including `.class` or `#id` selectors (or any of the combound selectors described below):

```
/* can also include class or id selectors */
/* this rule applies to h2 elements, "menu" classed elements, and the
   #sidebar element */
h2, .menu, #sidebar {
    background-color: gray;
}
```

Note that since later rules earlier ones, you can use a group select to apply a property to multiple different element, but then include additional rules to add variations. For example, you can have one rule that applies "general" styling to a large class of elements, with further rules then customizing particular elements.

```
/* all headings are Helvetica, bold, and purple */
h1, h2, h3 {
    font-family: Helvetica;
    font-weight: bold;
    color: #4b2e83; /* UW purple */
}

/* h2 elements are not bolded, but italic */
h2 {
    font-weight: normal; /* not bold, overrides previous rule */
    font-style: italic;
}
```

Combined Selectors

It is also possible to combine element, class, and id selectors together to be more specific about where a rule applies. You do this by simply putting the class or id selector *immediately after* the previous selector, without a comma or space or anything between them:

```
/* Selects only p elements that have class="alert"
   Other p elements and "alert" classed elements not affected */
p.alert {
  color: red;
}

/* Selects only h1 elements that have id="title" */
/* Note that this is redundant, since only one element can have the id! */
h1#title {
  color: purple;
}
```

```
/* Selects elements that have class "alert" AND class "success" */
.alert.success {
  color: green;
  font-size: larger;
}

/* And can combine with group selector */
/* applies to <p class="highlighted"> and <li class="selected"> */
p.highlighted, li.marked {
  background-color: yellow;
}
```

This specificity can allow you to reuse class names (e.g., for shared semantics and readability purposes) but have them work differently for different elements. So a "highlighted" paragraph `p.highlighted` might look different than a "highlighted" heading `h1.highlighted`.

Note that putting a space between the selectors parts instead specifies a **descendant selector**, which has a totally separate meaning. Every character matters!

Descendant Selector

So far, all selectors mentioned will apply to matching elements regardless of where they are in the HTML element tree. But sometimes you want to be more specific and style only a set of elements that exist within a particular parent or ancestor element, and not all the other matching elements elsewhere in the page. You can do this form of targeted selecting using a **descendant selector**. This is written by putting a blank space ( ) between selectors. Elements are only selected if they have *parents that match the selectors that precede them*:

```
<header>
   <h1>Welcome to the page</h1>
   <p>I am a special paragraph</p>
</header>
<section>
   <p>some other paragraph</p>
</section>
```

```
/*
  Selects p elements that exist within header elements
  Other p elements will not be affected
 */
header p {
    /* ... */
```

```
}
```

You can have as many "levels" of a descendant selector as you want, and each level can be made up of any kind of selector. However, it is best to not have more than 2 or 3 levels. If you need to be more specific than that, then perhaps defining a new `.class` is in order.

```
/* selects elements with class="logo"
   contained within <p> elements
   contained within <header> elements */
header p .logo {
    /* ... */
}
```

Note that descendant selectors will select matching descendant elements *anywhere* lower in the tree branch, not just direct children, so the `.logo` elements here could be nested several layers below the `<p>` element (perhaps inside a `<span>`). This is usually a good idea because you may introduce new nesting layers to your page as you go along, and don't want to modify the CSS. But if you really want to select only *direct* children, you can use a variant known as a **child selector**, which uses a `>` symbol to indicate direct descendants only:

```
<body>
  <p>Body content</p>
  <section>
    <p>Section content</p>
  </section>
</body>
```

```
/* Selects page content (immediately within body),
   not section content (immediately within section) */
body > p {
  color: blue;
}
```

Pseudo-classes

The last kind of selector you will commonly use in web development is the application of what are called **pseudo-classes**. These select elements based on what **state** the element is in: for example, whether a link has been visited, or whether the mouse is hovering over some content. You can almost think of these as pre-defined classes built into the browser, that are added and removed as you interact with the page.

Pseudo-classes are written by placing a colon (`:`) and the name of the pseudo-class immediately after a basic selector like an element selector. You'll see this most commonly with styling hyperlinks:

```
/* style for unvisited links */
a:link { /*...*/ }

/* style for visited links */
a:visited { /*...*/ }

/* style for links the user is hovering over with the mouse */
a:hover { /*...*/ }

/* style for links that have keyboard focus */
a:focus { /*...*/ }

/* style for links as they are being 'activated' (clicked) */
a:active { /*...*/ }
```

Remember to always set both `hover` *and* `focus`, to support accessibility for people who cannot use a mouse. Additionally, `a:hover` *must* come after `a:link` and `a:visited`, and `a:active` must come after `a:hover` for these states to work correctly.

Note that there are many additional pseudo-classes, including ones that consider specific element attributes (e.g., if a checkbox is `:checked`) or where an element is located within its parent (e.g., if it is the `:first` or `:last-child`, which can be useful for styling lists).

Attribute Selectors

Finally, it is also possible to select element that have a particular attribute by using an **attribute selector**. Attribute selectors are written by placing brackets `[]` after a basic selector; inside the brackets you list the attribute and value you want to select for using `attribute=value` syntax:

```
/* select all p elements whose "lang=sp" */
p[lang="sp"] {
    color: red;
    background-color: orange;
}
```

It is also possible to select attributes that only "partially" match a particular value; see the documentation for details.

Note that it is most common to use this selector when styling form inputs; for example, to make checked boxes appear different than unchecked boxes:

```
/* select <input type="checked"> that have the "checked" state */
input[type=checkbox]:checked {
```

```
    color: green;
}
```

6.2 Property Values

This section of the guide provides further details about the possible *values* that may be assigned to properties in CSS rules. These specifics are often relevant for multiple different properties.

Units & Sizes

Many CSS properties affect the **size** of things on the screen, whether this is the height of the text (the `font-size`) or the width of a box (the `width`; see the next chapter). In CSS, you can use a variety of different **units** to specify sizes.

CSS uses the following **absolute units**, which are the same no matter where they are used on the page (though they are dependent on the OS and display).

Unit	Meaning
`px`	**pixels** ($\frac{1}{96}$ of an inch, even on high-dpi "retina" displays)
`in`	**inches** (OS and display dependent, but maps to physical pixels in some way)
`cm`, `mm`	centimeters or millimeters, respectively
`pt`	points (defined as $\frac{1}{72}$ of an inch)

Although technically based on `in` as a standard, it is considered best practice to always use `px` for values with absolute units.

CSS also uses the following **relative units**, which will produce sizes based on (relative to) the size of other elements:

Unit	Meaning
`em`	Relative to the **current** element's font-size. Although originally a typographic measurement, this unit will **not** change based on `font-family`.
`%`	Relative to the **parent** element's font-size *or* dimension. For font-size, use `em` instead (e.g., `1.5em` is `150%` the parent font-size).
`rem`	Relative to the **root** element's font-size (i.e., the `font-size` of the root `html` or `body` element). This will often be more consistent than `em`.

Unit	Meaning
`vw`, `vh`	Relative to the **viewport** (e.g., the browser window). Represents 1% of the viewport width and height, respectively. This unit is not supported by older browsers.

Note that most browsers have a default font size of `16px`, so `1em` and `1rem` will both be initially equivalent to `16px`.

In general, you should specify font sizes using *relative units* (e.g., `em`)—this will support accessibility, as vision-impaired users will be able to increase the default font-size of the browser and all your text will adjust appropriately. Absolute units are best for things that do not scale across devices (e.g., image sizes, or the maximum width of content). However, using relative sizes will allow those components to scale with the rest of the page.

- Font-sizes should always be relative; layout dimensions may be absolute (but relative units are best).

Colors

Colors of CSS properties can be specified in a few different ways.

You can use one of a list of 140 predefined color names:

```
p {
   color: mediumpurple;
}
```

While this does not offer a lot of flexibility, they can act as useful placeholders and starting points for design. The list of CSS color names also has a fascinating history.

Alternatively, you can specify a color as a "red-green-blue" (RGB) value. This is a way of representing *additive color*, or the color that results when the specified amount of red, green, and blue light are aimed at a white background. RGB values are the most common way of specifying color in computer programs.

```
p {
   color: rgb(147, 112, 219); /* medium purple */
}
```

This value option is actually a *function* that takes a couple of parameters representing the amount of red, green, and blue respectively. Each parameter ranges from `0` (none of that color) to `255` (that color at full). Thus `rgb(255,0,0)` is pure bright red, `rgb(255,0,255)` is full red and blue but no green (creating magenta), `rgb(0,0,0)` is black and `rgb(255,255,255)` is white.

Note that if you want to make the color somewhat transparent, you can also specify an alpha value using the `rgba()` function. This function takes a 4th parameter, which is a decimal value from `0` (fully transparent) to `1.0` (fully opaque):

```
p {
   background-color: rgba(0,0,0,0.5); /* semi-transparent black */
}
```

CSS also supports `hsl()` and `hsla()` functions for specifying color in terms of a hue, saturation, lightness color model.

Finally, and most commonly, you can specify the RGB value as a hexadecimal (base-16) number.

```
p {
   color: #9370db; /* medium purple */
}
```

In this format, the color starts with a `#`, the first two characters represent the red (ranging from `00` to `FF`, which is hex for `255`), the second two characters represent the green, and the third two the blue:

Figure 6.1: How to reading a hex value, from Smashing Magazine.

This is a more compact and efficient way to describe the RGB of a color, and is how most digital artists convey color. See this article for more details about encoding colors.

Fonts and Icons

As mentioned in the previous chapter, you specify the typographical "font" for text using the **`font-family`** property. This property takes a *comma-separated list* of fonts to use; the browser will render the text using the first font in the list that is available (you can think of the rest as "back-ups").

```
p {
    font-family: 'Helvetica Nue', 'Helvetica', 'Arial', sans-serif;
}
```

The last font in the list should always be a generic family name. These are a list of "categories" that the browser can draw upon even if the computer doesn't have any common fonts available. In pracice, the most common generic families used are `serif` (fonts with serifs, e.g., "Times"), `sans-serif` (fonts *without* serifs, e.g., "Arial"), and `monospace` (fonts with equal width characters, e.g., "Courier").

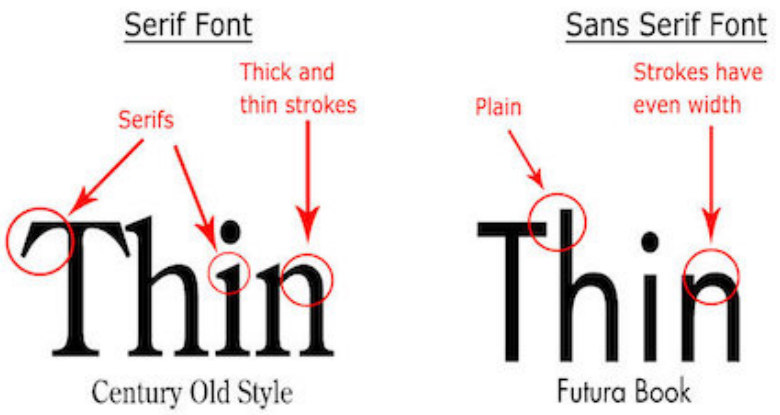

Figure 6.2: Serif vs. Sans-Serif fonts. From 99designs.ca.

It is also possible to include specific fonts in your web page, which will be delivered to the browser by the web server in case the client doesn't have the previously available. You do this manually by using the `@font-face` rule, and specifying the url for the font file (usually in `.woff2` format).

However, it is usually easier to instead include a stylesheet that *already has this rule in place*. For example, the Google Fonts collection provides more than 800 different freely available fonts that you can include directly in your web page:

```
<head>
    <!-- ... -->

    <!-- load stylesheet with font first so it is available -->
    <link href="https://fonts.googleapis.com/css?family=Encode+Sans" rel="stylesheet">

    <!-- load stylesheet next -->
    <link href="css/style.css" rel="stylesheet">
</head>
```

```
body {
    font-family: 'Encode Sans', sans-serif; /* can now use Encode Sans */
}
```

Notice that the `<link>` reference can be to an external file on a different domain! This is common practice when utilizing fonts and CSS frameworks.

Important Note that when using Google Fonts, you'll need to specify if you also want variations such as **bold** or *italic* typesets. For example, the Encode Sans font is available in font weights (what you would set with `font-weight`) from `100` to `900`, but you need to specify these in the linked resource:

```
<!-- includes normal (400) and bold (700) weights -->
<link href="https://fonts.googleapis.com/css?family=Encode+Sans:400,700" rel="styles
```

If you don't include the different set of glyphs for the bolded font, then setting the text in that font to bold won't have any effect (because the browser doesn't now how to show text in "Bold Encode Sans")!

You can also use external font styles to easily add **icons** (symbols) to your web page by using an *dingbat (icon) font*. These are similar in concept to the infamous Wingdings font: they are fonts where instead of the letter "A" looking like two bent lines with a bar between, it instead looks like a heart or a face or some other symbol. By including such a font, you get access to a large number of symbols that can easily be styled (read: colored and sized) to suit your needs.

Emoji are defined using a different set of Unicode values, and are browser and operating-system instead of being available through a font.

The most popular icon set is called Font Awesome. You can include the Font Awesome font by linking to it as any other stylesheet:

```
<link href="https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.
```

- This will load the stylesheet from a Content Delivery Network (CDN), which is a web service intended to quickly serve files commonly used by multiple websites. In particular, browsers will *cache* (save) copies of these files locally, so that when you visit a different website (or the same website for a second time), the file will already be downloaded. We will load most of the CSS and JavaScript frameworks used in this class from a CDN.

  The `.min` before the `.css` extension in the filename is a convention to indicate that the file is *minimized*: all extraneous spaces, comments, etc. have been removed to make the file as small and quick to download as possible.

Font Awesome icons can then be included in your HTML by including an element with an appropriate CSS class (that sets the style of that content to be the correct font with the correct "character" content). For example, you can include a Universal Access icon with

```
<i class="fa fa-universal-access" aria-hidden="true"></i>
```

- Because the icon has no semantic meaning beyond an appearance, it is often included with a semantically-deficient (empty) `<i>` element. Remember to include an `aria-hidden` attribute so that a screen-reader won't try to read the strange letter!

Other icon-based fonts include Gylphicons and Octicons.

Backgrounds and Images

You have previously seen how to use the the `background-color` property to color the background of a particular element. However, CSS supports a much wider list of background-related properties. For example, the `background-image` property will allow you to set an image as the background of an element:

```
header {
    background-image: url('../img/page-banner.png');
}
```

This property takes as a value a `url()` data type, which is written like a function whose parameter is a string with the URI of the resource to load. These URIs can be absolute (e.g., `http://...`), or relative **to the location of the stylesheet** (not to the web page!—you may need to use `..` if your `.css` file is inside a `css/` directory.

There are additional properties used to customize background images, including where it should positioned in the element (e.g., centered), how large the image should be, whether it should repeat, whether it should scroll with the page, etc.

```
header {
    background-image: url('../img/page-banner.png');
    background-position: center top; /* align to center top */
    background-size: cover; /* stretch so element is filled; preserves ratio (img may be cropp
    background-repeat: no-repeat; /* don't repeat */
    background-attachment: fixed; /* stay still when window scrolls */
    background-color: beige; /* can still have this for anything the image doesn't cover
                                (or for transparent images) */
}
```

This is a lot of properties! To understand all the options for their values, read through the documentation and examples (also here and here).

To try and make things easer, CSS also includes a **shorthand property** called just **`background`**. Shorthand properties allow you to specify multiple properties at once, in order to keep your code more compact (if somewhat less readable).

Shorthand properties values are written as a *space-separated list of values*; for example, the above is equivalent to:

```
header {
    background: url('../img/page-banner.png') top center / cover no-repeat fixed bei
}
```

- The `background-position` and `background-size` are separated by a `/` since they both can have more than one value.
- You can include some or all of the available background properties in the shorthand. Unlike most shorthand properties, the `background` properties can go in any order (though the above is recommended).

Note that a shorthand property is interpreted as writing out all of the rules it replaces; so will *replace* any previous properties within the same rule:

```
body {
    background-color: gold;
    background: mediumpurple; /* later property override previous ones */
                              /* page will have a purple background */
}
```

Additionally, all of the `background` properties support multiple backgrounds by using a *comma-separated* list of values. This can enable you to easily overlay partially-transparent images on top of each other, similar to using layers in Photoshop.

There are many, many other properties you can use as well to style your page. Be sure and look through all the documentation and examples, and explore the source code of existing pages to see how they achieved particular effects!

Resources

- CSS Diner a fun game for practicing with different CSS selectors
- CSS Units and Values (MDN)
- The Code Side of Color
- CSS Backgrounds (MDN)

Chapter 7

CSS Layouts

The previous chapters have discussed how to use CSS to specify the appearance of individual html elements (e.g., text size, color, backgrounds, etc). This chapter details how to use CSS to declare where HTML elements should *appear* on a web page!

7.1 Block vs. Inline

Without any CSS, html elements follow a default **flow** on the page based on the order they appear in the HTML. Layout is based on whether the element is a *block element* or an *inline element*.

As mentioned in Chapter 3, **inline** elements (e.g., `<em>`, `<a>`, `<span>`) are put next to each other on a single line (left to right, unless you specify a right-to-left language). **Block** elements (`<p>`, `<ul>`, `<div>`) are placed on subsequent "lines", from top to bottom.

```
<div>Block element</div>
<div>Block element</div>
```

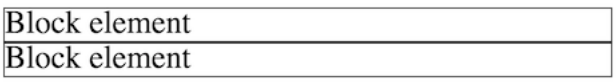

Figure 7.1: Example of block elements, placed on top of each other.

```
<span>Inline element</span>
<span>Other inline element</span>
```

Figure 7.2: Example of inline elements, next to each other (even if the code is on separate lines).

However, you can force an element to be either `block` or `inline` by declaring the **`display`** CSS property:

```
.inlined {
  display: inline;
}
```

```
<!-- this will produce the same result as using <span> elements -->
<div class="inlined">Inline element</div>
<div class="inlined">Other inline element</div>
```

The `display` property also allows you to remove elements from the page flow entirely:

```
.hidden {
  /* do not show element at all */
  display: none;
}
```

There are several other more specific settings, the most useful of which is `inline-block`. This lets you treat an element as `inline` (no line break after), but also gives you the ability to specify its `width`, `height`, and other properties only adjustable on `block` elements. This is particularly useful for making things like lists (`<ul>`) appear "inline".

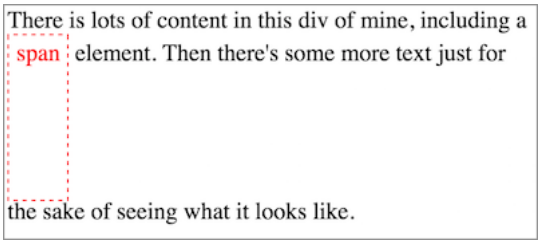

Figure 7.3: An example of an element with `display:inline-block`.

7.2 The Box Model

The **CSS Box Model** is one of the core concepts in CSS, and is one of the central frameworks by which elements are visually laid out on the page.

All HTML element (*including text!*) include an imaginary **box** around their content. Elements are laid out with their boxes next to each other (side by side for `inline`, stacked on top for `block`). These boxes are normally just large enough to contain the content inside the element, but you can use CSS to alter the size of and spacing between these boxes in order to influence the layout.

First off, you can set the **`width`** and **`height`** of elements explicitly, though be careful when you do this. If your `width` and `height` are too small, the element's content will be clipped by default (a behavior controlled by the `overflow` property). It's generally best to set only the width **or** the height, but not both. You can also specify a `min-width` or `min-height` to ensure that the width or height is at least a particular size. Conversely, you can use `max-width` and `max-height` to constrain the size of the element.

In order to adjust the spacing between boxes, you can manipulate one of 3 properties:

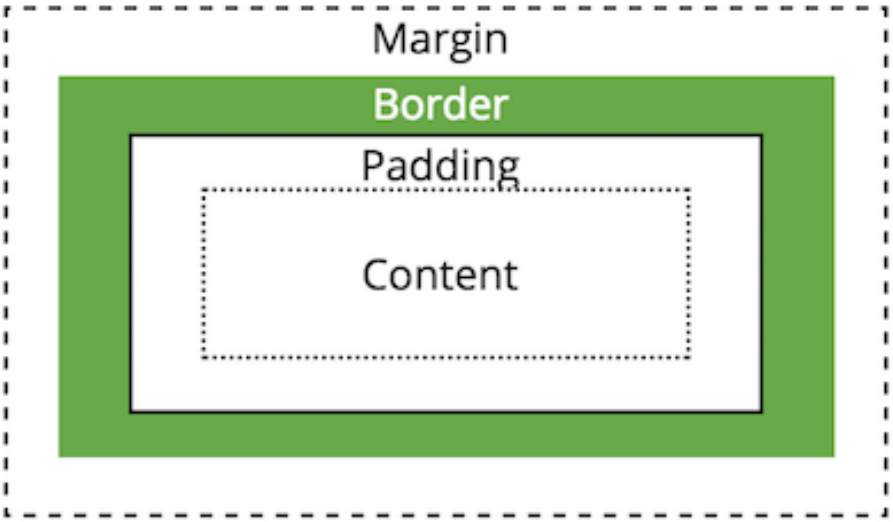

Figure 7.4: A diagram of the box model properties

Padding

The **padding** is the space between the content and the border (e.g., the edge of the box).

It is possible to specify the padding of each side of the box individually, or a uniform padding for the entire element:

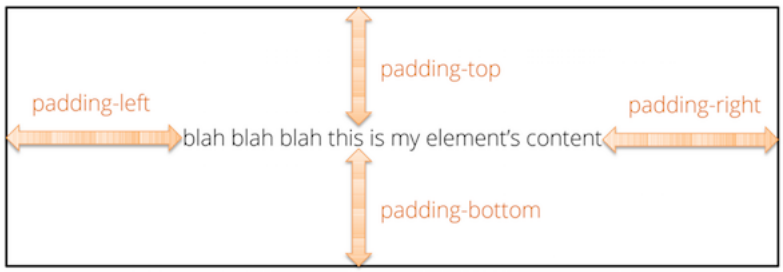

Figure 7.5: An element's padding.

```
/* specify each side individually */
div {
  padding-top: 1em;
  padding-bottom: 1em;
  padding-left: 2em;
  padding-right: 0; /* no units needed on 0 */
}

/* specify one value for all sides at once */
div {
  padding: 1.5em;
}

/* specify one value for top/bottom (first)
   and one for left/right (second) */
div {
  padding: 1em 2em;
}
```

Border

The **border** (edge of the box) can be made visible and styled in terms of its width, color, and "style", listed in that order:

```
.boxed {
   border: 2px dashed black; /* border on all sides */
}

.underlined {
   border-bottom: 1px solid red; /* border one side */
}
```

```
.something { /* control border properties separately */
   border-top-width: 4px;
   border-top-color: blue;
   border-top-style: dotted;
   border-radius: 4px; /* rounded corners! */
}
```

Margin

Finally, the **margin** specifies the space *between* this box and other nearby boxes. `margin` is declared in an equivalent manner to `padding`.

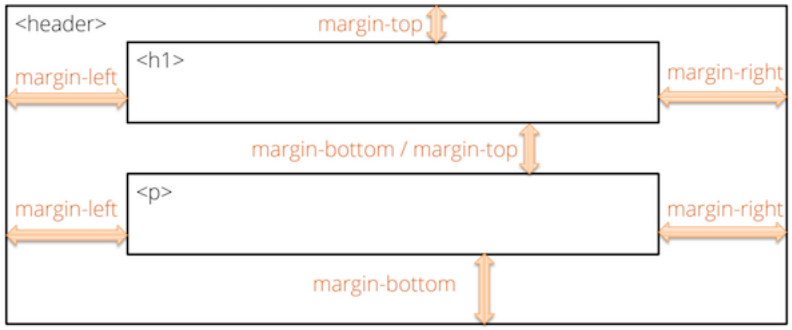

Figure 7.6: An element's margins.

Note that browsers typically collapse (overlap) the margins of adjacent elements. For example, if you have two paragraphs on top of one another, and you set `margin-bottom` on the first and `margin-top` on the second, most browsers will overlap those margins and just use the larger of the two values to determine the spacing.

Box-Sizing

An elements `padding`, `border`, and `margin` can be used to put space between element content on the page. However, when you assign an explicit **`width`** or **`height`** to an element, the dimension you specify **does not include** the padding or border when calculating the size of the element on the page! That is, if you have an element with the properties

```
.my-box {
    width: 100px;
    padding: 10px; /* includes both left and right */
}
```

Then the element will take up 120px on the screen: the width plus the left and right padding.

However, when specifying more complex or responsive layouts, it's often useful to have `width` represent the entire width of the box, and not need to account for the border and padding separately in calculations. You can do this using the `box-sizing` property—a value of `border-box` will indicate that specified *size* of the box (e.g., the `width`) should include the size of the padding and border when determining the content area.

It's common to want to apply this property to **all** of the elements on the page, which you can do with the `*` selector (like a wildcard from the command-line!):

```
* {
    box-sizing: border-box; /* all elements include border and padding in size */
}
```

This is a common enough change that you may wish to include it in *all* of your `.css` files!

7.3 Changing the Flow

Specifying the `display` style and **box** properties will adjust the layout of HTML elements, but they are still following the browser's default *flow*. The layout rules will still apply, and elements are influenced by the amount of content and the size of the browser (e.g., for when inline elements "wrap").

However, it is possible to position elements outside of the normal flow by specifying the `position` property.

For example, giving an element a **`position:fixed`** property will specify a "fixed" position of the element *relative to the browser window*. It will no move no matter where it appears in the HTML or where the browser scrolls. See this Code Pen for an example.

- In order to specify the location for a **`fixed`** element, use the `top`, `left`, `bottom`, and/or `right` properties to specify distance from the appropriate edge of the browser window:

```
/* make the <nav> element fixed at the top of the browser window */
nav {
    position: fixed;
    top: 0;  /* 0px from the top */
    left: 0; /* 0px from the left */
    width: 100%; /* same as parent, useful for spanning the page */
}
```

You can also specify an element's `position` to be **`relative`**, meaning *relative to its normal position*. Note that this leaves the element within normal flow (e.g., for how its padding effects other elements around it). See this Code Pen to explore this option.

Finally, you can specify an element's `position` to be **`absolute`**, meaning *relative to it (positioned) parent element*. If the parent has not been explicitly positioned (has a declared `position` property), the element is positioned relative to the root element. Try it out in this Code Pen. This mostly remove an element from normal flow, though its parent may still be part of the flow and thus may influence the absolutely positioned element's location.

Floating

You can also remove an element from it's normal position in the *flow* by making it **float**. This is commonly done with pictures, but can be done with any element (such as `<div>`). A floated element is shoved over to one side of the screen, with the rest of the content wrapping around it:

```
.floating-image {
  float: right;
  margin: 1em; /* for spacing */
}
```

Content will continue to sit along side a floated element until it "clears" it (gets past it to the next line). You can also force content to "clear" a float by using the `clear` property. An element with this property *cannot* have other elements floating to the indicated side:

```
.clear-float {
    clear: both; /* do not allow floating elements on either side */
}
```

The `float` property is good for when you simply want some content to sit off to the side. But you should **not** try to use this property for more complex layouts (e.g., multi-column structures); there are better solutions for that.

7.4 Flexbox

The `position` and `float` properties allow you to have individual elements break out of the normal page flow. While it is possible to combine these to produce complex effects such as **multi-column layouts**, this approach is fraught with peril and bugs due to browser inconsistencies. In response, CSS3 has introduced new standards specifically designed for non-linear layouts called **Flexbox**. The Flexbox layout allows you to efficiently lay out elements inside a container (e.g.,

columns inside a page) so that the space is *flexibly* distributed. This provides additional advantages such as ensuring that columns have matching heights.

Flexbox is a new standard that is now supported by most modern browsers; it has a buggy implementation in Microsoft IE, but is supported in the standards-compliant Edge. For older browsers, you can instead rely on a grid system from one of the popular CSS Frameworks such as Bootstrap.

Despite it's capabilities, Flexbox still is designed primarily for one-directional flows (e.g., having one row of columns). To handle true grid-like layouts, browsers are adopting *another* emerging standard called **Grid**. The Grid framework shares some conceptual similarities to Flexbox (configuring child elements inside of a parent container), but utilizes a different set of properties. Learning one should make it easy to pick up the other. Note that the grid framework is less well supported than even Flexbox (it is not supported by IE, Edge, or common older Android devices), so should be used with caution.

To use a Flexbox layout, you need to style *two* different classes of is that you need to style *two* different classes of elements: a **container** (or **parent**) element that acts as a holder for the **item** (or **child**) elements—the child elements are *nested* inside of the parent:

```
<div class="flex-container"> <!-- Parent -->
  <div class="flex-item">Child 1</div>
  <div class="flex-item">Child 2</div>
  <div class="flex-item">Child 3</div>
</div>
```

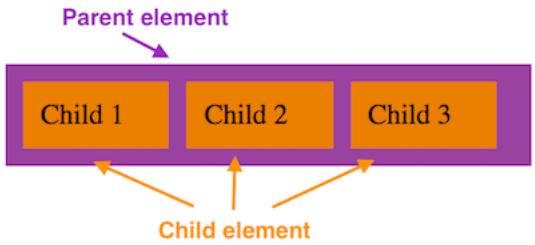

Figure 7.7: An example of a simple Flexbox layout.

Note that the "outer" parent element has one class (e.g., `flex-container`, but you can call it whatever you want), and the "inner" child elements share another (e.g., `flex-item`). Creating an effective Flexbox layout requires you to specify properties for *both* of these classes. You most often use `<div>` elements for the parent and child elements (and the child of course can have further content, including more divs, nested within it).

In order to use a Flexbox layout, give the *parent* element the **`display:flex`** property. This will cause the contents of that parent element to be layed out

according to a "flex flow":

```
.flex-container { /* my flexbox container class */
    display: flex;
}
```

A flex flow will lay out items *horizontally* (even if they are block elements!), though you can adjust this by specifying the *parent's* `flex-direction` property.

By default, a flex container will fill its parent element (the whole page if the container is in the `<body>`), and the child items will be sized based on their content like normal. While it is possible to then use dimensional properties such as `width` and `height` to size the children within the horizontal layout, Flexbox provides further options that make it more *flexible*.

Any *immediate child* of the flexbox container can use additional properties to define how that particular item should be layed out within the container. There are three main properties used by flex **items**:

```
* { box-sizing: border-box; } /* recommended for item sizing */

.flex-item {
    flex-grow: 1; /* get 1 share of extra space */
    flex-shrink: 0; /* do not shrink if items overflow container */
    flex-basis: 33%; /* take up 33% of parent initially */
}
```

- **`flex-grow`** specifies what "share" or ratio of any extra space in the container the item should take up. That is, if the container is `500px` wide, but the items' only takes up `400px` of space, this property determines how much of the remaining `100px` is given to the item.

  The value is a unitless number (e.g., `1` or `2`, defaulting to `0`), and the amount of remaining space is divided up *proportionally* among the items with a `flex-grow`. So an item with `flex-grow:2` will get twice as much of the remaining space as an item with `flex-grow:1`. If there are 4 items and `100px` of space remaining, giving each item `flex-grow:1` will cause each item to get `25px` (100/4) of the extra space. If one of the items has `flex-grow:2`, then it will get `40px` ($\frac{2}{1+1+1+2} = \frac{2}{5} = 40\%$) of the extra space, while the other three will only get `20px`.

  In practice, you can give each item a property `flex-grow:1` to have them take up an equal amount of space in the container.

- **`flex-shrink`** works similar to `flex-grow`, but in reverse. It takes as a value a number (default to `1`), which determine what "share" or ratio it should shrink by in order to accommodate any overflow space. If the specified dimensions of the items exceeds the dimensions of the container (e.g., 4 `100px` items in a `300px` container would have `100px` of "overflow"),

the `flex-shrink` factor indicates how much size needs to be "taken off" the item. A higher number indicates a greater amount of shrinkage.

In practice, you will often leave this property at default (by not specifying it), *except* when you want to make sure that an item does NOT shrink by giving it `flex-shrink:0`.

- **`flex-basis`** allows you to specify the "initial" dimensions of a particular item. This is similar in concept to the `width` property, except that `flex-basis` is more flexible (e.g., if you change the `flex-direction` you don't also have to change from `width` to `height`). Note that this value can be an dimensional measurement (absolute units like `100px`, or a relative unit like `25%`).

  In practice, using percentages for the `flex-basis` will let you easily size the columns of your layout.

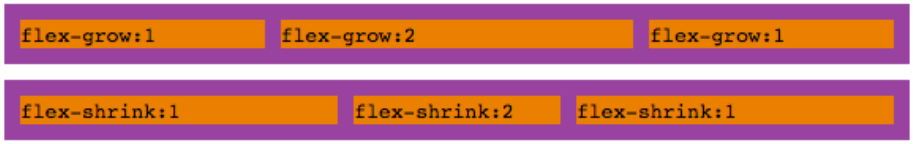

Figure 7.8: Top: visual example of `flex-grow`. Bottom: visual example of `flex-shrink`. Notice how much extra "space" each item has after the text content.

There is also a *shortcut property* `flex` that allows you to specify all three values at once: give the `flex-grow`, `flex-shrink`, and `flex-basis` values separated by spaces (the second two being optional if you want to use the default values).

The Flexbox framework also provides a number of additional properties that you can specify on the **container** to customize how items of different sizes are organized on the screen:

- `justify-content` specifies how the items should be spread out across the container. Note that items that have `flex-grow:1` will use up the extra space, making this less relevant.
- `align-items` is used to specify "cross-axis" alignment (e.g., the vertical alignment of items for a horizontal row).
- `flex-wrap` is used to have items "wrap around" to the next line when they overflow the container *instead of* shrinking to fit. You can then use the `align-content` property to specify how these "rows" should be aligned within the container.

While it may seem like a lot of options, Flexbox layouts will allow you to easily create layouts (such as multi-column pages) that are otherwise very difficult

with the regular box model. Moreover, these layouts will be flexible, and can easily be made **responsive** for different devices and screen sizes.

Resources

- The Box Model (MDN)
- The CSS Box Model (CSS-Tricks)
- A Complete Guide to Flexbox The best explanation of Flexbox properties you'll find.
- A Complete Guide to Grid A similar explanation, but for the Grid framework (not discussed here).

Chapter 8

Responsive CSS

These days the majority of people accessing any web site you build will be using a device with a small screen, such as a mobile phone. But phones come in a wide range of sizes and resolutions, and many people will still access that same site from a laptop or desktop with a much larger monitor (as well as other capabilities, such as a mouse instead of a touchscreen). Different screens may require different visual appearances: for example, a three-column layout would be hard to read on a mobile phone! This poses and interesting *design* dilemma: how do you build one site that looks good and works well on both tiny phones and gigantic desktop monitors?

The modern solution to this problem is **Responsive Web Design**, which involves using CSS to specify *flexible* layouts that will adjust to the size of the display: content can be layed out one column on a small mobile screen, but three columns on a large desktop.

This chapter discusses CSS techniques used to create **responsive** web sites. These techniques underlie popular CSS frameworks, so it is important to understand them even if you rely on such tools.

8.1 Mobile-First Design

Responsive design is often framed as a technique to "make it *also* work on mobile". This approach feels easy since websites are usually developed and tested on desktops, and follows from the software principle of *graceful degradation* (systems should maintain functionality as portions break down, such as the "capability" of having screen real estate).

But since websites are more likely to be visited on mobile devices, a better approach is to instead utilize **mobile-first design**. This is the idea that you

Figure 8.1: Example of a responsive web site on multiple devices, with content and layout changing at each size. From responsivedesign.is.

should develop a website so it content and purpose is effectively presented on mobile devices (e.g., the most restricted in terms of screen size, capabilities, etc). Only once this base level of functionality is in place should you add features to make it *also* look good on larger devices such as desktops. This approach is also known as *progressive enhancement*: provide the core functionality, and then add "extra" features as more capabilities become available.

Rather than viewing mobile devices as "losing" features, look at desktops as "gaining" features! This will help you to focus on better supporting more common mobile devices.

Remember: working on your personal machine doesn't ensure that it will work on anyone else's!

Mobile-First Design Principles

While there is no magic formula for designing websites to support mobile devices, there are a few general principles you should follow:

- **Layout**: On mobile devices, blocks of content should stack on top of each other, rather than sitting side by side in columns—mobile devices want to only scroll on one axis. `fixed` content should be kept to a minimum, as it reduces the amount of scrollable screen real-estate.

As you gain more screen space on desktops, you will *want* to break content up into columns or otherwise constrain its width so that it doesn't stretch to ridiculous lengths; this helps with readability.

- **Media**: Small screens don't have enough space to necessitate very large, high-resolution images and video. Moreover, large images have large file sizes, and so will take a long time to download on slow mobile connections (not to mention eating away at limited data plans). Use compressed or lower-resolution images on mobile, and consider using background colors or linear gradient fills instead of background images. You can use higher-resolution media (and more of it!) on desktops, which usually have higher bandwidth available for downloading.
- **Fonts**: Make sure to use a large enough font that it is readable on small screens… but don't make headings or callout text *too* large so that you lose that precious real estate! You can make them more styled and prominent on desktop, where there is room for such flourishes. Be sure to use relative units to accommodate mobile user preferences and screen size variation. Also remember that special web fonts you may be downloading will also take up extra bandwidth!
- **Navigation**: Site navigation links take up a lot of room on small screens and may end up wrapping to multiple lines. Use small tab bars, or menu icons (e.g., the "hamburger icon") to show complex menus on command. Most CSS frameworks provide some kind of collapsible navigation for mobile devices.
- **Input and Interaction**: Tap/click targets need to be large-enough on mobile to select using a finger, especially for people with poor eyesight or thick fingers. Tiny icons placed right next to one another, or one-word hyperlinks are difficult to select accurately. Specifying a data type on form fields (e.g., email address, phone number, date, integer) also generates optimized on-screen keyboards, making data entry much easier.
- **Content**: For some sites, you may even want to adjust what content is shown to mobile users as opposed to desktop users. For example, a phone number might become a large telephone icon with a `tel:` hyperlink on mobile phones, but simply appear as a normal telephone number on desktop displays.

Specifying Viewport

Mobile web browsers will do some work on their own to adjust the web page in response to screen size—primarily by "shrinking" the content to fit. This often produces the effect of the website being "zoomed out" and the user enlarging the web page to a readable size and then scrolling around the page to view the content. While it may "work" it is not an ideal user interaction—this behavior

can also interfere with attempts to be explicit about how webpages should adjust to the size of the screen.

To fix this, you need to specify the viewport size and scale by including an appropriate `<meta>` element in your HTML:

```
<head>
  <meta charset="utf-8"> <!-- always need this -->
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=

  <!-- more head elements, including <link> ... -->
</head>
```

Including this element will keep the text size from adjusting to the browser's width (though *very* narrow browsers may still shrink the text).

While not technically part of any web standard, the `viewport` meta element was introduced by Safari and is supported by most mobile browsers. The `content` attribute for the above meta tag sets 3 properties for mobile browsers: `width` (how big the viewport should be, specified to be the size of the device screen), `initial-scale` (how much to initially "zoom" the page, specified to be 1x or no zoom), and `shrink-to-fit` (tells Safari 9.0+ not to shrink the content to fit).

You should include the `viewport` meta tag in all of your responsive pages. Make it part of your default HTML template!

8.2 Media Queries

In order to define a webpage with a responsive appearance, you need to be able to *conditionally* change the applied style rules depending on the size of the screen (or browser window). We can conditionally apply CSS rules by using **media queries**. A *media query* is a bit like an `if` statement in CSS: it specifies a *condition* and the rules (*selectors* and *properties*) that should apply when that condition is true.

Media queries have syntax similar to the following:

```
/* A normal CSS rule, will apply to all screen sizes */
body {
    font-size: 14px;
}

/* A Media Query */
@media (min-width: 768px)
{
    /* these rules apply ONLY on screens 768px and wider */
```

```
    /* a normal CSS rule */
    body {
        font-size: 18px;
        background-color: beige;
    }
    /* another CSS rule */
    .mobile-call-icon {
        display: none; /* don't show on large displays */
    }
}
```

A media query is structurally similar to a normal CSS rule. The "selector" is written as **`@media`**, indicating that this is a media query not a normal CSS rule. The `@media` is then followed by a *query expression*, somewhat similar to the boolean expressions used in `if` statements. Expressions are written in parentheses, with the media feature to check, followed by a colon (**`:`**), followed by the value to check against. There are no *relational operators* (no `>` or `<`) in media queries, so you use media features with names such as `min-width` and `max-width` (to represent `width > x` and `width < x` respectively).

- Media feature comparisons are not strict inequalities, so `min-width: 1000px` can be read as "width greater than or equal to 1000px".
- It is also possible to use the *logical operators* `and` and `not` to combine media feature checks. You can produce an "or" operator using a group selector (a comma **`,`**).

```
/* style rules for screens between 768px and 992px */
@media (min-width: 768px) and (max-width: 992px) { }

/* style rules for screens larger than 700px OR in landscape orientation */
@media (min-width: 700px), (orientation: landscape) { }
```

The `@media` rule is followed by a **`{}`** block, inside of which are listed *further regular CSS rules*. These rules will *only* be applied if the `@media` rule holds. If the `@media` rule does not apply, then these "inner" rules will be ignored. These "inner" rules can utilize all the selectors and properties value outside of media queries—think of them as mini "conditional" stylesheets!

You need to put *full rules* (including the selector!) inside of the media query's body. You can't just put a property, because the browser won't know what elements to apply that property to.

Media queries follow the same ordering behavior as other CSS rules: **the last rule on the page wins**. In practice, this means that media queries can be used to specify conditional rules that will *override* more "general rules". So in the example above, the page is set to have a `font-size` of `14px` using a rule that will apply on any screens. However, on larger screens, the media query will

also apply, overriding that property to instead make the default font size `18px`.

Following a **mobile-first approach**, this means that your "normal" CSS should define the styling for a the page on a mobile device. Media queries can then be used to add successive sets of rules that will "replace" the mobile styling with properties specific to larger displays.

```
/* on small mobile devices, the header has a purple background */
header {
    font-size: 1.2rem;
    background-color: mediumpurple;
}

/* on 768px OR LARGER displays */
@media (min-width: 768px) {
    header {
        font-size: 1.5rem; /* make the header larger font on larger displays */
    }
}

/* on 992px OR LARGER displays */
@media (min-width: 992px) {
    header {
        background-image: url('../img/banner.png') /* use background image */
    }
}
```

- In this example, the `<header>` is given a simple purple background and default font size. When loaded on mobile devices, this is the only rule that applies (the media queries aren't valid), so that is all the styling that occurs.
- But on devices `768px` *or wider* (like a tablet), the first media query is activated. This will then run a second rule that applies to the `<header>`, overriding the font to be larger (`1.5rem`)—it's as if we had listed those two `header` rules one after another, and the later one wins. But the first rule continues to apply, making the background purple.
- Finally, a device `992px` *or wider* (like a desktop computer), will cause *both* of the media queries to execute. (since a device whose width is greater than `992px` is ALSO greater than `768px`). Thus the `<header>` will be given a purple background and default font size, which will then be overridden to be a larger font by the first media query. The second media query will add an additional property (a banner image background), which will combine with the previous purple background (e.g., if the banner has any transparency). So on large displays, there will be a banner background on top of purple, with text in a larger (`1.5rem`) font.

This structure (starting with "default" mobile rules and then using media queries

with *increasingly larger* `min-width` values) produces an effective mobile-first approach and clean way of organizing how the appearance will change as the screen gets larger. Note that while you can define as many media queries as you want, most professionals define only a few that match the common breakpoints between phone, tablet, and desktop screen widths. Since the media queries need to come **after** the mobile rules, they are often included at the end of the stylesheet—give all the mobile rules first, then all the media queries that modify them. (You can and should put multiple rules in each media query).

(Alternatively, it's also reasonable to organize your stylesheets based on page "section", with the mobile rules for that section given before the media queries for that section. E.g., after all your rules for creating the page header, put the media query with variations for desktop headers. Use whatever organization makes your code readable and maintainable, and leave plenty of comments to guide the reader!)

8.2.1 Example: Responsive Flexbox

As another example of using media queries to produce a responsive website, consider how they can combine with Flexbox to produce a single-column layout on mobile devices, but a multi-column layout on larger displays. Because a Flexbox layout is just a property applied to existing elements, we can effectively "turn on" the `flex` layout by using media queries.

Consider some simple HTML:

```
<div class="row">
  <div class="column">column ONE content</div>
  <div class="column">column TWO content</div>
  <div class="column">column THREE content</div>
  <div class="column">column FOUR content</div>
</div>
```

By default (without any style rules applied to the `.row` or `.column` classes), the four inner `<div>` elements (as block elements) will stack on top of each other. This is the behavior you want on mobile narrow screens, so now additional CSS or Flexbox usage is needed.

In order to turn these divs into columns on larger displays, introduce a media query that applies the `flex` layout to the `.row` (which acts as the *container*), thereby lining up the `.column` elements (which act as the *items*)

```
/* on devices 768px OR WIDER */
@media (min-width: 768px)
{
    .row { /* row becomes a flexbox container */
        display: flex;
```

```
    }

    .column  { /* column becomes a flexbox item */
        flex-grow: 1; /* make the columns grow equally to fill the row */
    }
}
```

You can also add an additional media query at another breakpoint so that the layout starts out stacked, then switches to two columns on medium-sized displays (leading to a two-by-two grid), and *then* switches to four columns on large displays:

```
/* on devices 768px OR WIDER */
@media (min-width: 768px)
{
    .row { /* row is a flexbox container */
        display: flex;
        flex-wrap: wrap; /* wrap extra items to the next "line" */
    }

    .column  { /* column is a flexbox item */
        flex-basis: 50%; /* columns take up 50% of parent by default */
        flex-grow: 1;
    }
}

@media (min-width: 1200px) {
    .column {
        flex-basis: auto; /* columns are automatically sized based on content */
    }
}
```

In this case, the first media query (`768px +` or medium-sized displays) applies the `flex` layout and specifies that the items should `wrap` if they overflow the container… which they will, since each item has a default `flex-basis` size of `50%` of the container. This will cause each of the 4 items to take up 50% of the parent, wrapping around to the next line.

Then when the screen is larger (`1200px +` or large-size displays), the second media query is applied and *overrides* the `flex-basis` so that it will automatically calculate based on the content size, rather than being `50%` of the parent. That way as long as the columns fit within the parent, they will all line up in a row (they have the `flex-grow` property to make them equally spread out).

The best way to get a feel for how this works is to see it in action: see **this CodePen** for an example of the above behavior—resize the browser and watch the layout change!

Media rules are a powerful and declarative way to create a single page that looks great on everything from a small mobile touchscreen to a large desktop monitor with a mouse, and form the foundation for responsive CSS frameworks that can help you easily create fantastic looking pages.

Resources

- Responsive Web Design Basics (Google)
- Using media queries (MDN)

Chapter 9

CSS Frameworks

Coming soon...