Client-Side Web Development

Joel Ross and Mike Freeman

November 25, 2018

Contents

About the Book

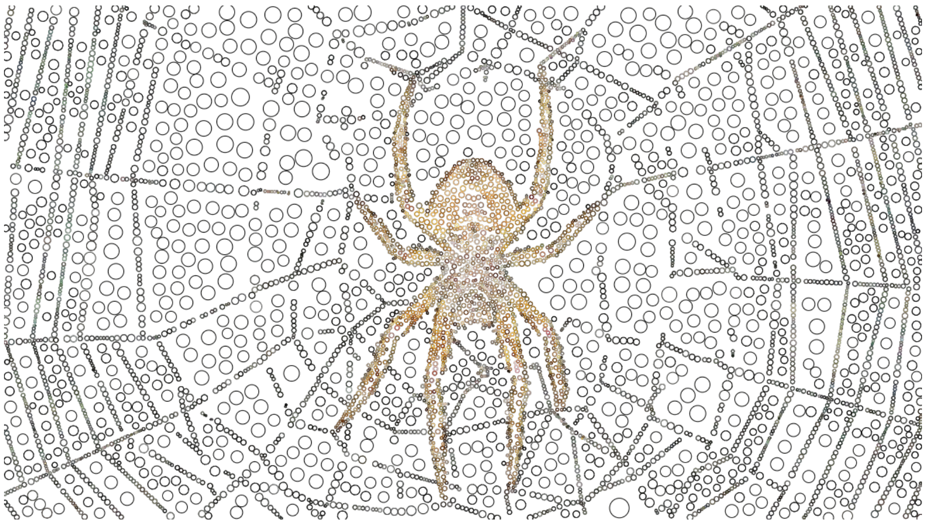


Figure 1:

About the 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 340: 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.

In summary, you'll need to install and set up the following software on your machine (more information below on each of these). Feel free to use alternative software, but these are the suggested programs for the course (i.e., the ones which we will provide support for):

- **Google Chrome** as your web browser
- **Visual Studio Code** as your code editor
- **Node.js** as a JavaScript engine
- **`npm`** as a JavaScript package manager (comes with Node.js)
- **`git`** as a version control system (for Windows users; already installed on Macs)
- A Bash terminal (e.g., **Git Bash** for Windows, or the **subsystem for Linux**; Terminal is already installed on Macs)
- **`live-server`** as a local development server (simply run `npm install live-server`)

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 pretty much always want to have these tools open when doing web development, especially when including interactivity via JavaScript.

Other modern browsers such as **Firefox** or **Microsoft Edge** will also function in this course 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 one of our 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 dialog 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.

While we suggest using VSCode for this course, other editors are also acceptable and may be of interest to you.

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. The document you are reading was authored in Atom.

Brackets

Brackets is a coding editor created 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 a great option 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. *Alternatively*, the 64-bit Windows 10 Anniversary update (August 2016) does include a version of an integrated Bash shell. You can access this by enabling the **subsystem for Linux** and then running `bash` in the command prompt.

- Note that Windows does come with its own command-prompt, called the *Command Prompt* (previously the *DOS Prompt*), but it has a different set of commands and features. *Powershell* is a more powerful version of the Command 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.

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 the Git Bash command shell.

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, Git Branches, and Git 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 need *at least* the current Long Term Support (LTS) version **8.11**, so you should update Node if you installed it previously (e.g., by downloading and reinstalling it). 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
```

Be sure and check that you have a recent enough 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 725,000 different packages.

If you have an older version of Node and `npm` (before Node 8.11 or `npm` 5.6), you will need to update them. You can re-install the latest version of Node, or just upgrade `npm` using the command **`npm install npm@latest -g`**. See below for an explanation of this command.

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!

`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 the packages listed under `"dependencies"` (as well as under `"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` automatically add them to the dependencies list by using the `--save` option:

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

will install `lodash` locally, and add it to 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).

If you want to use a *globally* installed package in your local project (e.g., have it be a dependency but not have to download and install it again), you can use `npm link` to "include" the global package locally. For example, the below will allow you to use a globally installed version of `lodash` so you don't have to download a copy for the project:

```
npm link lodash
```

Be sure to `link` any global packages *before* you run `npm install` so you don't download any packages from `package.json` that you already have!

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. Note that you will generally need to use one package manager or other; don't try to mix them in a single project!

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, the 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.

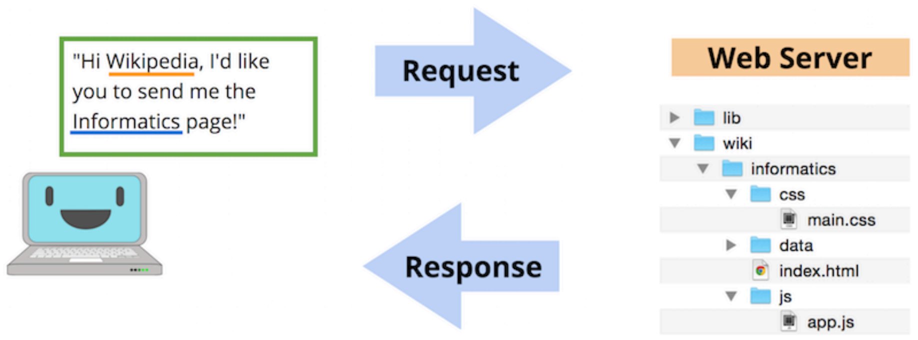


Figure 2.1: A diagram of client/server communication.

The web server will receive this request, and based on its content (e.g., the parameters of the URL) 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 responding 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 will 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 are interpreted by the *browser*, and are 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", since the code is run on the local machine).

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:

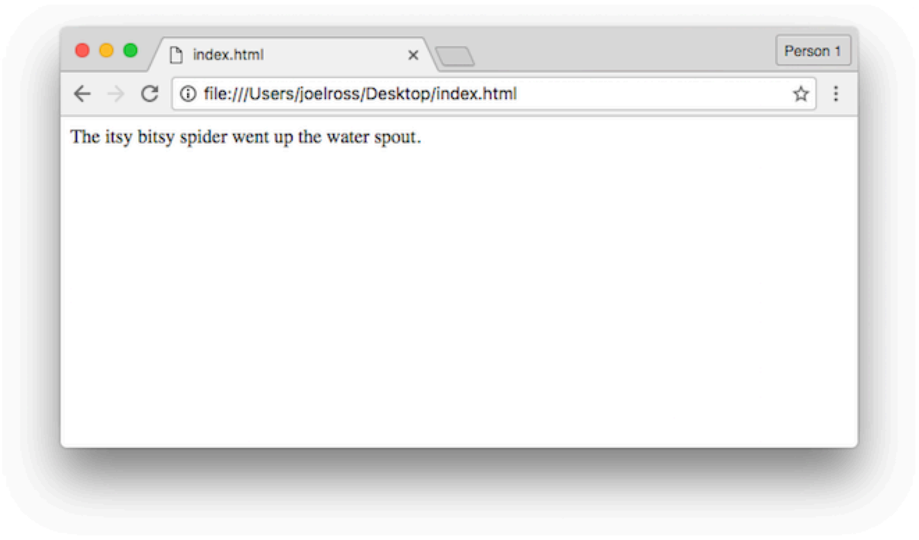
- **`.html`** files containing code written in HTML (HyperText Markup Language). This code will specify the textual and *semantic* content of the web page. See the chapter HTML Fundamentals for details on HTML.
- **`.css`** files containing code written in CSS (Cascading Style Sheets). This code is used 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):



The itsy bitsy spider went up the water spout.

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.

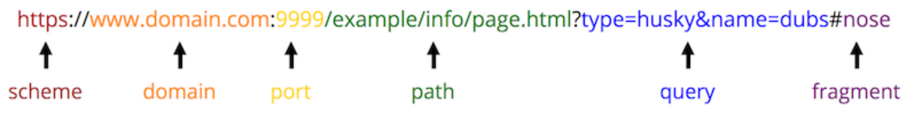

https://www.domain.com:9999/example/info/page.html?type=husky&name=dubs#nose
scheme
domain
port
path
query
fragment

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` (secure HTTP). *Don't use insecure* `http`*!*

  Web page *hyperlinks* often include URIs with the `mailto` protocol for email links, or the `tel` protocol 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, but for most web URIs this would be the address (e.g., `google.com` or `ischool.uw.edu`).

- **`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 on your computer. 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 specify a path to a folder rather than a file (including `/` as the "root" folder), most web servers will serve the file named `index.html` from that folder (i.e., the path "defaults" to `index.html`). As such, this is the traditional name for the HTML file containing a website's 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 fragment.

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 many 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` or `https` 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 of it as 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 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 the Markdown markup language, 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 fast 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 understand 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**:

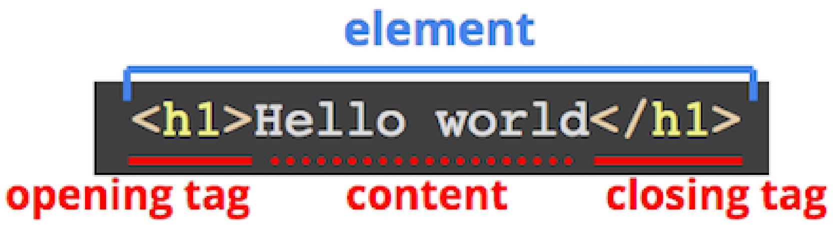


Figure 3.1: Basic syntax for an HTML element.

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) are 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>
```

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

Nevertheless, when writing HTML code, use line breaks and spacing for readability (to make it clear what content is part of what element).

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 Elements

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 (one followed by a line break)
- `<span>`: a span (section) of content. Also acts as an empty *inline* element (one 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:

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

The contents of the comment tag can span multiple lines, so you can comment multiple lines by surrounding them all with a single `<!--` and `-->`.

Because the comment 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, 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` (no spaces are allowed around the `=`); 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` (**src**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 visibility 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 it a unique identifier so that you can refer to it 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="#title">Link to element on current page with `id="title"`</a>
```

Note that, when specifying an id (i.e., `<h1 id="title">`) you do not include the `#` symbol. However, to *link* to an element with id `title`, you include the `#` symbol before the id (i.e., `<a href="#title">`)

- 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, you leave off the end tag entirely:

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

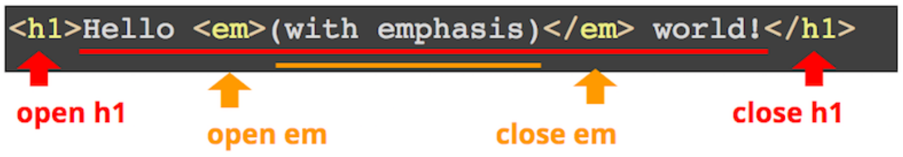
Older versions of HTML (and current related languages like XML) required you to include a forward slash `/` just before the ending greater-than symbol. This "closing" slash indicated that the element was complete and expected no further content:

```
<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):



```
<h1>Hello <em>(with emphasis)</em> world!</h1>
```

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
<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 12 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.

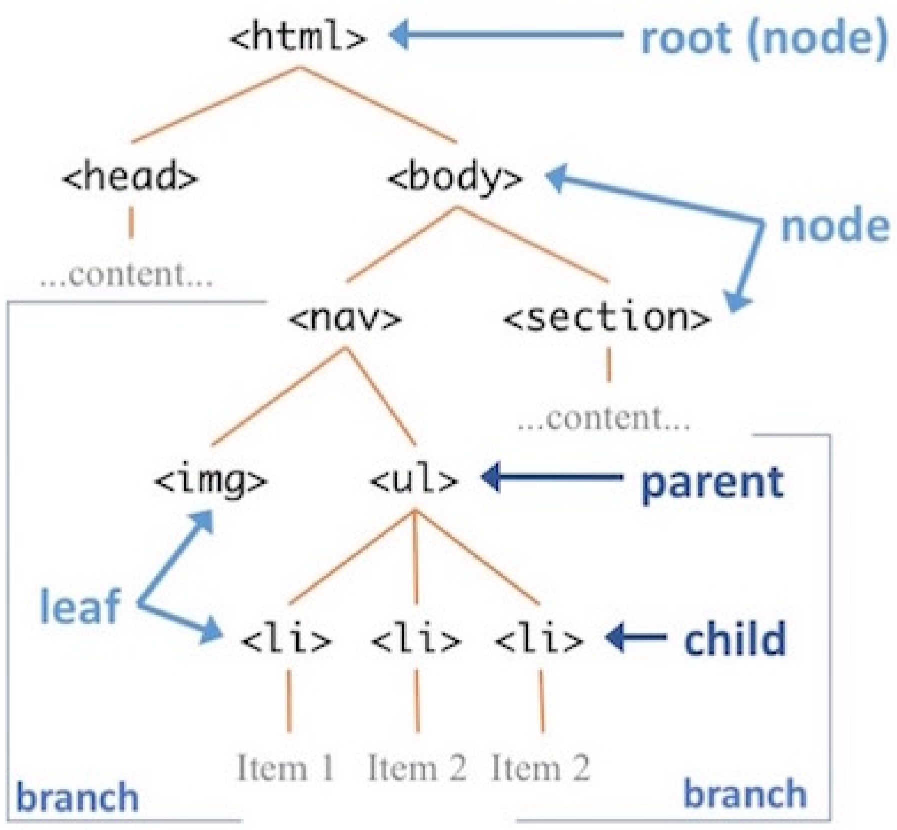


Figure 3.3: An example DOM tree (a tree of HTML elements).

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.

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

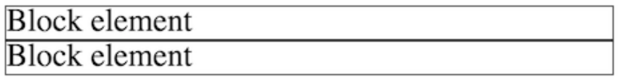
  

  Figure 3.4: Two block elements rendered on a page.

  A `<div>` ("division") is the most basic block element: it represents a block of content but provides no other semantic meaning.

- **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>`).

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

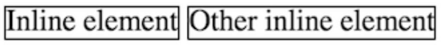
  

  Figure 3.5: Two inline elements rendered on a page.

  A `<span>` is the most basic inline element: it represents an inline span of content but provides no other semantic meaning.

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 considered invalid markup.

Each element has a different default display type (e.g., `block` or `inline`, but it is also possible to change how an element is displayed using CSS. See Chapter

7.

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:

```html
<!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:

  ```html
  <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 is 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!

- Alternatively, you can use the VS Code shortcut of writing an exclamation point (`!`) then hitting the `tab` key in a `.html` file to create a similar page skeleton.

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

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. It will provide detailed information on the following strategies for building accessible pages:

- Using **semantically meaningful** HTML tags
- Providing **alternate text** for visual features (i.e., images)
- Describing the **role** of elements on your page (if it isn't clear based on the element type)
- Hiding **unnecessary content** (i.e., visual flare) from screen readers

4.1 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.*

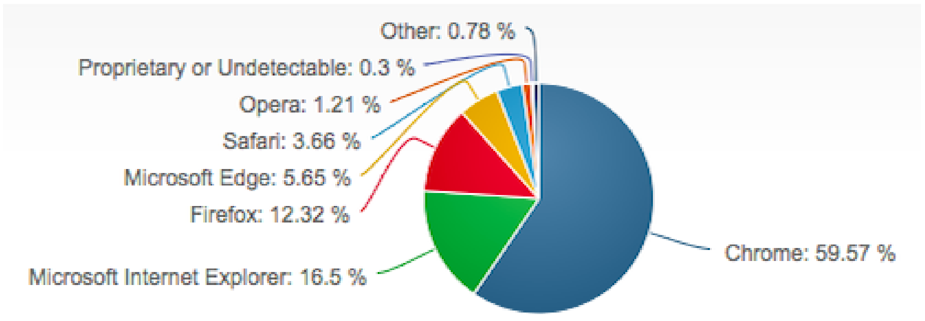


Figure 4.1: *Desktop* browser market share July 2017, from netmarketshare.com. See also caniuse.com's usage table for information on mobile and older browsers.

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 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 the `hyphens` property (for controlling word breaks), while `-ms-hyphens` would use the IE version. This practice is currently discouraged and being phased 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 (and so don't ned to test it on every possible browser!). Luckily, 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 page's 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!

4.2 Why Accessibility

Consider the following hypothetical webpage user:

> Tracy is a 19-year-old college student and was born blind. Through high school she did well as she could relying 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 site. Indeed, even web companies with their capitalist world-view 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. This fact is increasingly being acknowledged by companies as key to usability, and thus it is important that you apply it to your own design work.

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).

4.3 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 documents 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 interfaces 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 that can be **perceived** by screen readers (so is not just presented visually).

Don't worry: there are simple, specific implementation steps you can use 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
<!-- html -->
<h1>This is a real heading!</h1>
<div class="fake-header">
    This just LOOKS like a heading.
</div>
```

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

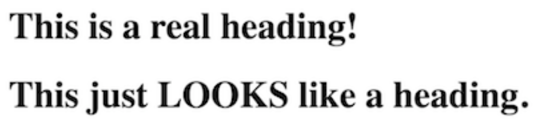


Figure 4.2: A real heading and a fake heading

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 **A**ccessible **R**ich **I**nternet **A**pplications 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:

```html
<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 (that is: use 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 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 ensure 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 nesting 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>`! A *heading* is element (e.g., `<h1>`) that includes a title or subtitle. A *<header>* 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).

A typical web page body may thus have a structure similar to:

```html
<!-- Starting from the body tag -->
<body>
    <!-- Page header -->
    <header>
        <nav>
            navigation links in here
        </nav>
    </header>

    <!-- Main page content -->
    <main>
        <section>
            content...
        </section>
        <section>
            content...
        </section>
    </main>

    <!-- Page footer -->
    <footer>
        copyright information here
    </footer>
</body>
```

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 with a magnifying glass icon). 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):

```html
<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.

```html
<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.

```html
<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 img 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 an ARIA attribute to provide an equivalent to an `alt` attribute 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)
- Using ARIA (w3c) official guidance for using the ARIA specification
- WAVE Accessibility Evaluation Service

Chapter 5

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 content, 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 chapters

5.1 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 dealing with 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.), without changing the content.
- 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, you 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*.

5.2 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 a 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.

- It is also possible to include CSS code directly in your HTML by embedded it in a `<style>` tag in the `<head>`, but this is considered bad practice (keep concerns separated!) and should only be used for quick tests.

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 chapters.

- `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 it's common practice to quote any specific font name as well (e.g., `'Arial'`).

  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 Chapter 7). 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 will 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 to 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 you 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 or snake_cased.

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>Mooooooooooooooooooooo.</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>
```

```
/* Style 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 of an id selector*, unless you are referring to a single, specific element.

5.3 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 applies 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 from the same HTML page, the files are processed *in order* as they are linked in the HTML. In processing a CSS file, 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**.

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

```html
<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:

```css
/* css */
.alert { color: red; }
div { color: blue; }
```

```html
<!-- html -->
<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 compound selectors described in Chapter 6 to be able to create reusable rules and 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 6

CSS Options

Chapter 5 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:

```css
/* 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 take precedence earlier ones, you can use a group select to apply a property to multiple different elements, 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 paragraph "Body content" (immediate child of body),
   not paragraph "Section content" (immediate child of 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).

nth-* Selectors

Two powerful pseudo-class selectors are the `nth-child` and `nth-of-type` selectors. These selectors allow you to select specific elements based on their position in the DOM. For example, you may want the third `<li>` element in a (each) list to have a certain style. To do that, you can use the either the `nth-child` or `nth-of-type` selector (there'a great description of the difference between these here):

```
/* Get third li element  */
li:nth-of-type(3) {
  color:red;
}
```

The `nth-*` selectors also support the keywords `odd` and `even`, which is really useful for styling tables:

```
/* Give every-other row a light-gray background */
tr:nth-of-type(even) {
  background-color:#eee;
}
```

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:

```css
/* 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:

```css
/* select <input type="checkbox"> 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)

Unit	Meaning
`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`.
`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:

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.

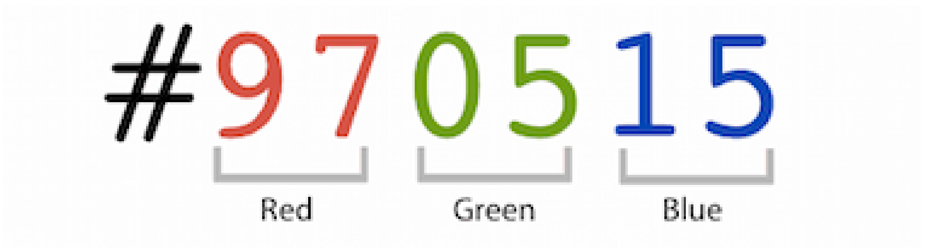


Figure 6.1: How to reading a hex value, from Smashing Magazine.

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”).

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>
```

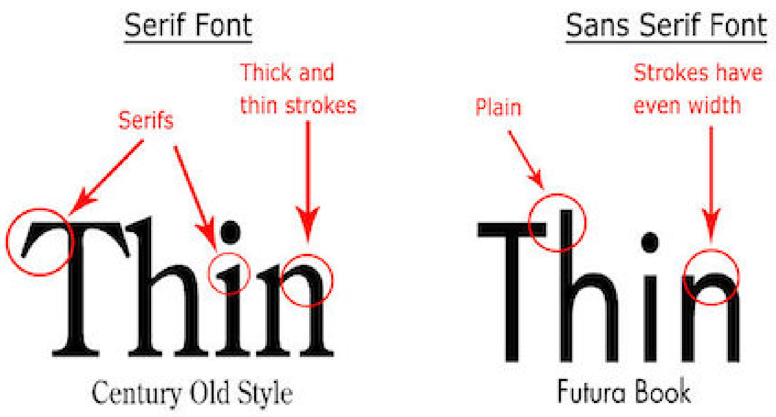


Figure 6.2: Serif vs. Sans-Serif fonts. From 99designs.ca.

```css
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:

```html
<!-- 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.css" rel="
```

- 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 ma
    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 c
                                (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!Placing elements exactly how you want them can be surprisingly difficult. Elements are arranged from the **top left corner** of the page, and are arranged based on three related (but distinct) concepts: - **Display**: determines how elements share horizontal space - **Position**: allows you to adjust the location of the natural flow of elements - **Box-model**: describes the amount of 2D space taken up by an element, and how much space is between elements

7.1 Display

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>
```

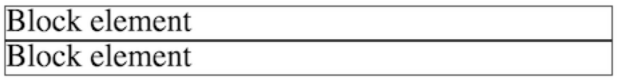
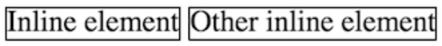


Figure 7.1: Example of block elements, placed on top of each other.

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



Inline element Other inline element

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:

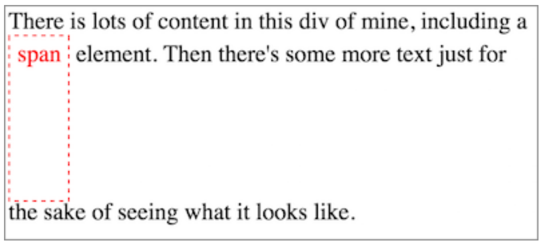
```css
.inlined {
  display: inline;
}
```

```html
<!-- 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:

```css
.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".



There is lots of content in this div of mine, including a span element. Then there's some more text just for the sake of seeing what it looks like.

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 elements (*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:

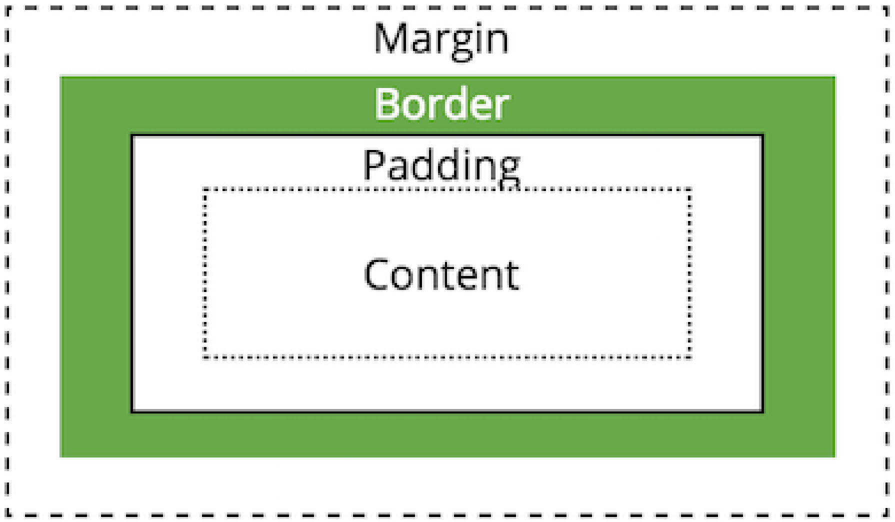


Figure 7.4: A diagram of the box model properties

Padding

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

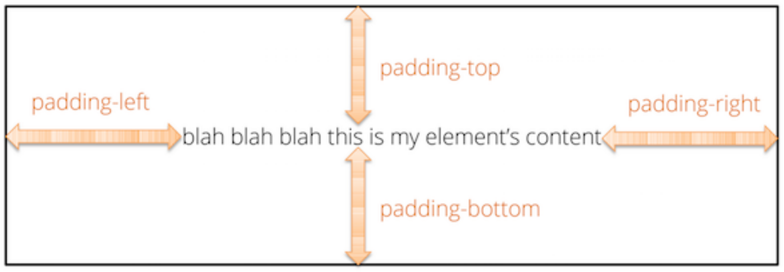


Figure 7.5: An element's padding.

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

```
/* 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`.

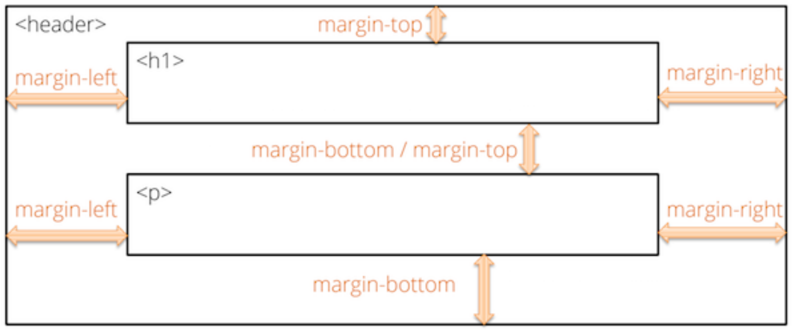


Figure 7.6: An element's margins.

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 element's `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 Position

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. Depending on the `position` property, you can shift the location of elements using the `top`, `bottom`, `left`, and `right` properties.There are four values that the `position` property can be set to:

> **Static**: By default, elements are positioned statically. This is their natural layout, and elements *will not* be shifted by the `top`, `bottom`, `left`, or `right` properties.
>
> **Relative**: Allows you to shift the element *relative* to it's natural position on the page. This enables you to move the element using the `top`, `bottom`, `left`, and `right` properties.

> **Fixed**: The *fixed* position allows you place an element in a consistent location within a browser window. For example, if you wanted a link to *always* be 50px from the bottom, you could set the properties: `position:fixed;` and `bottom:50px;`.
>
> **Absolute**: The *absolute* position allows you to place an element a specific number of pixels from it's (first non-static) parent element. This allows you to specify where *within a parent element* you want an element to be positioned.

Note, in order to get `position:absolute;` to work *"as expected"*, you'll need to specify that the *parent element* is `position:relative;`. See this post for more information.

As an 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 */
  }
  ```

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 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>
```

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

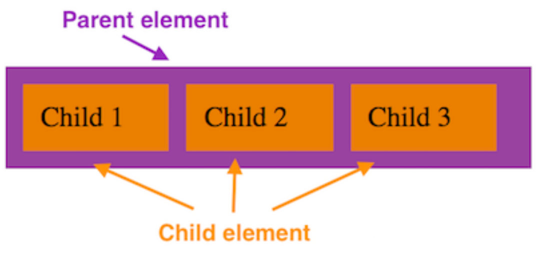


Figure 7.7: An example of a simple Flexbox layout.

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.

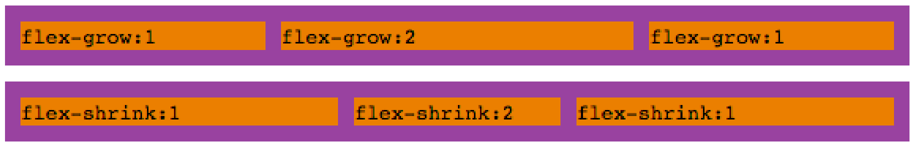


Figure 7.8: Top: visual example of `flex-grow`. Bottom: visual example of `flex-shrink`. Notice how much extra "space" each item has after the text content.

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).

Note that if unspecified, the `flex-basis` property will be set to `0`, rather than the `auto` value it has if unspecified.

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

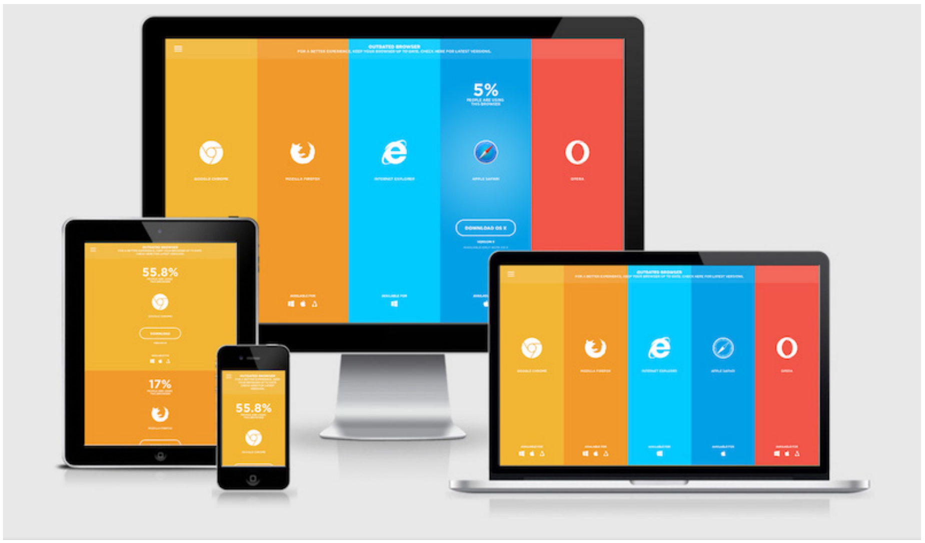


Figure 8.1: Example of a responsive web site on multiple devices, with content and layout changing at each size. From responsivedesign.is.

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!

A great way to test the responsiveness of your design is to click on the **Toggle Device Toolbar** icon in your element inspector:

This will allow you to specify a variety of different screensizes for specific devices of interest (i.e., an iPhone 5s, etc.).

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:

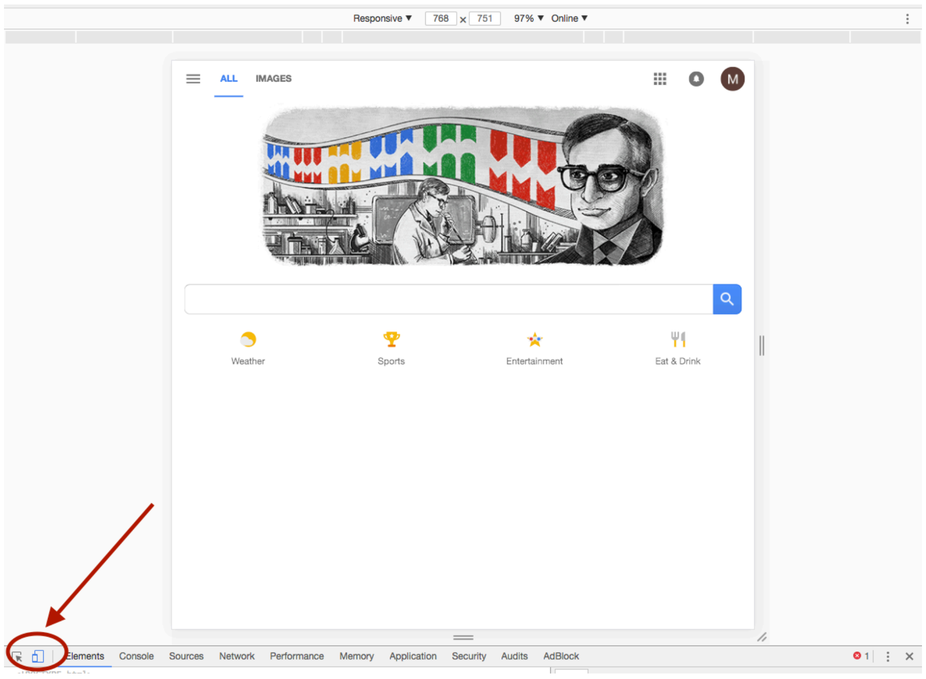


Figure 8.2: toggle device toolbar

- **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.

  Page bloat is a real problem. You don't need huge images… or possibly any images at all!

- **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:

```html
<head>
  <meta charset="utf-8"> <!-- always need this -->
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <!-- 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:

```css
/* 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.

```css
/* 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 don't apply to the given screensize), 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!)

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:

```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*)

```css
/* 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:

```css
/* 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

The previous chapters have covered how to use CSS to make your page both stylish and responsive. While it is certain possible to implement an entire site by creating your own CSS rules, that can quickly get tedious: often you'd like a page to have a "standard" set of rules (because the browser's style-less appearance) and then customize those rules to your liking.

For this reason, most professional web developers utilize an existing **CSS Framework** instead. A CSS Framework is a stylesheet (a `.css` file!) that contains a large list of pre-defined rules that you can apply to your page. CSS frameworks provide a number of benefits:

- *Applies attractive default styling to all HTML elements*: CSS frameworks make your pages instantly look better through a bunch of element selector rules. Frameworks provide pleasant default fonts, line spacing, spacing, and link styling without any extra effort on your part.
- *Provides style classes for common UI components*: framework stylesheets will also include CSS classes you can add to your markup to easily include badges, in-page tabs, drop-down buttons, multi-column layouts, and more. Frameworks enable you to add style your page by specifying CSS classes, rather than needing to define multiple CSS rules for a single effect.

CSS frameworks are thus designed to make your life easier and your development more efficient—while still enabling you to provide your own customizations and styling with all of the power of CSS.

There are many different CSS frameworks to choose from. Some popular ones include:

- **Bootstrap** is the most commonly used CSS framework on the web. Sometimes called "Twitter Bootstrap", it was originally created at Twitter to enforce some consistency among their internal tools, but was released as an open-source project in 2011. Its popularity has benefits and drawbacks:

it's very well tested, documented, and supported, but it's also so prevalent that it's default look has become cliché.

The latest release of Bootstrap, v4 was released in 2017. This version includes an update look and feel, and makes it relatively easy to create your own customized version. Bootstrap 4 also uses Flexbox as a foundation for its grid system, offering some better performance. For this and other reasons, Bootstrap 4 does **not support** IE 9 or earlier browsers. For older browsers, you'd want to use Bootstrap 3.

This text discusses Bootstrap 4 as an example. Be careful that you are looking at information about the right version if you seek out documentation or help online!

- **Foundation** is Bootstrap's chief rival, and has a reputation for being more ahead-of-the-curve than Bootstrap (introducing new features sooner). For example, it was the first framework to use a responsive mobile-first design, and provided a Flexbox-based grid long before Bootstrap. Foundation has most of the same UI elements as Bootstrap, but with a different look and feel (that can be customized through a web-based tool).

- **Material Components for the Web (MCW)** is an official implementation of Google's Material Design visual language. This is the look-and-feel found in most Google products and Android applications. Material Design is very opinionated so MCW is very difficult to customize. The MCW style class names are also very verbose, as they follow the Block, Element, Modifier (BEM) naming scheme.

- **Materialize** is the other popular Material Design implementation. This is an open-source project, so it is not provided nor supported by Google. However, it is structurally similar to Bootstrap, making it easy to learn and popular among people who know that framework.

- **normalize.css** is not so much a full framework as a small utility that performs browser normalization (also called a "reset"). `normalize.css` standardizes the quirks across browsers and making it so that the same HTML and CSS is displayed more consistently and without errors. Note that most other frameworks include some version of this package (for example, Bootstrap contains a fork called Reboot).

  If you choose **not** to use a framework, you should still include `normalize` to make your sites consistent!

Other frameworks exist as well, each with its own benefits and drawbacks (e.g., Skeleton is designed to be as lightweight as possible, but at the cost of features).

9.1 Using a Framework

The important thing to realize is that a CSS framework is **just a stylesheet with a bunch of rules that someone else wrote for you**. There's nothing magic about them. You can look at the stylesheet and see all the rules that have been defined, and it's all stuff you could have written yourself (though some of it can be quite complex). But those rules have been crafted by professionals and tested on a wide array of browsers to ensure consistent results, so it's a good idea to build on top of them rather than trying to re-invent the wheel.

For example, you can view the `normalize.css` file at https://necolas.github.io/normalize.css/7.0.0/normalize.css and see that it looks exactly like the CSS you might write on your own!

Including a Framework

Because a CSS framework is *just a CSS file*, you include one in your page using a `<link>` element just like with any other stylesheet:

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

  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.1.0/css/bootstrap.css">
  <link rel="stylesheet" href="css/my-style.css">
</head>
```

Important! Note that you link the CSS framework *before* you link your own stylesheet. Remember that CSS is read top-to-bottom, and that the last rule on the page wins. By putting your stylesheet second, any rules you define will *override* the rules specified in the framework, thereby allowing you to continue to customize the appearance.

- For example, `bootstrap.css` gives the `<body>` a color of `#212529` (very dark gray); you may instead want to have a different default text color::

  ```
  /* my-style.css */
  body { color: blue; }
  ```

  If you included your stylesheet *first* in the `<head>`, then your rule would be applied initially but would be overridden by the `bootstrap.css` file, causing your rule to seem to have no effect! Thus you should **always** put your own custom CSS as the *last* `<link>` in the page, after any CSS frameworks.

It is most common to include the **minified** versions of CSS frameworks: these are usually named as **`.min.css`** files (e.g., `bootstrap.min.css`). A minified file is one that has extra comments and white space removed, creating a file with a much smaller size (remember that each space and line break is a character so takes up a byte! Bootstrap goes from 160 kilobytes to about 127 kilobytes when minimized). Because web pages have to be downloaded, smaller pages will be downloaded faster, and thus will be "quicker" for the user. While an unminimized CSS file won't "break the bank", the extra white space doesn't provide anything that you need.

- You can almost always switch between the minimized and non-minimized file just by changing the file name between `file.css` and `file.min.css` (when using a CDN).

There are a few different ways of accessing a CSS framework's stylesheet in order to link it into your page:

Linking to a CDN

The easiest option is usually to link to a Content Delivery Network (CDN), which is a web service intended to quickly serve files commonly used by multiple websites. To do this, you simply provide the URL for the file on the CDN's servers as the `href` of your link (as in the example above).

CDNJS is a CDN for packages and libraries operated by Cloudflare, and is a good place to start if you're unsure of the CDN link for a packages. Other packages may have different CDN hosting (e.g., Bootstrap recommends a link from its own BootstrapCDN).

In addition to being simple, CDNs also provide a distinct speed benefit: CDNs replicate their content to machines in several regions of the world, and use dynamic Domain Name Service (DNS) resolution to steer users to the machine nearest them. So a user in Australia might download the Bootstrap CSS from a server located in Singapore, while a user in France might get the same CSS file from a server located in Ireland. This helps increase the speed that the files are downloaded.

Additionally, CDNs allow a site to take advantage of **browser caching**: browsers will save previously downloaded versions of a file to avoid downloading it again. If multiple sites all use the CDN version of Bootstrap, then the browser only has to download that file the first time you visit one of those sites. With popular frameworks like Bootstrap, it's highly likely that your user has already visited a site that links to Bootstrap's CDN version, and thus the Bootstrap files are already in the user's browser cache.

Finally, CDNs allow the frameworks to be accessed as a service, meaning that developers may be able to quickly patch bugs (if the fix doesn't break existing code) without the developer having to do anything. (This feature can also be seen as a drawback, as you need to trust the library developer not to break anything or accidentally include malicious code).

The only real disadvantage to linking to a CDN is that it won't work when you are offline. If you commonly do your development offline, or if you are building a web application that is meant to run offline, you'll need to utilize a different method.

Downloading the Source

If you can't link to a CDN, you'll need to download the source file directly to your computer and saving it as a source code file in your project (often in a `lib/` folder). These files can be found on the framework's website, or often in their GitHub repo (look for the `.css` file, or for a `dist/` distribution folder).

In addition to being available offline, downloading CSS files directly provides a few other advantages:

- Frameworks may allow you to customize their content even before you include the file: selecting only the components you need, and adjusting base styling properties before downloading your customized source file.
- If you use a CSS pre-compiler such as SASS, you can often modify the source-code mix-ins to more easily provide your own customizations and override the default fonts, colors, sizes, etc. simply by changing some variables.
- If you use a build system such as gulp or webpack, then you can often combine *all* of your CSS files into a single file. This reduces the total number of files that need to be downloaded, which can increase the page load speed—especially on slower mobile networks.

Note that it is also possible to utilize a **package manager** such as `npm` to download and manage the source for frameworks. This has the advantage of enabling you to avoid adding large extraneous source files to your code repo, as well as managing any dependencies for those libraries.

Recall that you can install a package using `npm`, saving the dependency into your project's `package.json` file:

```
# install bootstrap
npm install --save bootstrap@4
```

(This explicitly loads Bootstrap 4. See below for details).

This will install the library's source code into your project's `node_modules/` folder. You will then need to dig around inside that source folder for the `.css` file you wish to link (it is in no way consistent for all frameworks)

```
<!-- notice the RELATIVE path to the file -->
<link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.min.css">
```

Because you've saved the dependency in your `package.json` file, you can use the `.gitignore` file to exclude the `node_modules/` folder from your code repo, having new developers install the required framework via `npm install`. You can also use `npm` to easily upgrade the packages (such as when Bootstrap 4 leaves beta).

Important: if you are publishing a site to GitHub Pages, you **must** add the `node_modules` folders to your code repo so that it will get uploaded to GitHub and the file will be present on the server! For this reason, I recommend just using a CDN when working with GitHub Pages.

9.2 Bootstrap

This section discusses *some* of the features and uses of the Bootstrap CSS framework. Note that many other frameworks are used in a similar fashion (though the specific classes and components may be different).

You include the Bootstrap framework by linking its `.css` file to your page (whether from a CDN or a local file). You can get the link for the latest version from the Bootstrap Home Page.

```
<link rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.0/css/bootstrap.min.css
      integrity="sha384-9gVQ4dYFwwWSjIDZnLEWnxCjeSWFphJiwGPXr1jddIhOegiu1FwO5qRGvFXO
      crossorigin="anonymous">
```

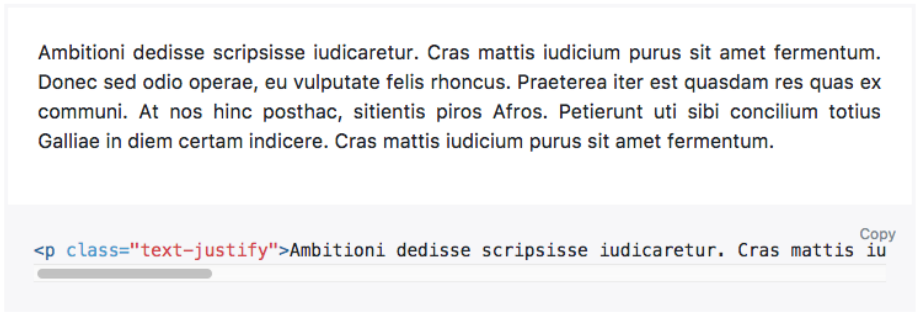
(This `<link>` example has 2 additional (optional) attributes: `integrity` gives a cryptographic hash of the source code so the browser can make sure you didn't download a malicious file by mistake, and `crossorigin` which says that no credentials are sent to the remote server—see the AJAX chapter for a more detailed discussion).

When you first include the Bootstrap framework, you'll immediately see the effects (usually for the better!): the default font has changed, margins are different, and so forth. The Bootstrap CSS file contains a number of rules that apply to elements without an extra work, including the `<body>` and headers. See the documentation for details about the default styling (as well as the entire "Content" section of the documentation).

Utility Classes

In addition to the base styling rules, Bootstrap also includes a huge number of rules that can be applied to elements by giving them a particular CSS **class**. These utility classes are what gives Bootstrap its power and flexibility.

For example, Bootstrap provides text utilities that allow you to style text just by giving it a CSS class, rather than needing to write a new rule for that element. This example shows the `text-justify` utility, which justifies the text (i.e., the letter- and word-spacing is adjusted so that the text falls flush with both margins):



Ambitioni dedisse scripsisse iudicaretur. Cras mattis iudicium purus sit amet fermentum. Donec sed odio operae, eu vulputate felis rhoncus. Praeterea iter est quasdam res quas ex communi. At nos hinc posthac, sitientis piros Afros. Petierunt uti sibi concilium totius Galliae in diem certam indicere. Cras mattis iudicium purus sit amet fermentum.

```
<p class="text-justify">Ambitioni dedisse scripsisse iudicaretur. Cras mattis iu
```

Figure 9.1: The `text-justify` utility class will justify a block of text. Example from the Bootstrap documentation.

While it would be possible to write such a rule yourself, Bootstrap provides these classes as "built-in", so you can focus on the semantics of the content rather than the CSS.

Many elements need to be given an additional utility class to indicate that they should use Bootstrap styling. For example, in order to give a `<button>` a Bootstrap style, it needs to have the class `btn`. This allows you to utilize Bootstrap's styling, but to not be forced to "override" that look and feel if you want to do something different.

```html
<button>Plain button</button>
<button class="btn">Bootstrap button</button>
<button class="btn btn-primary">Primary colored button</button>
```

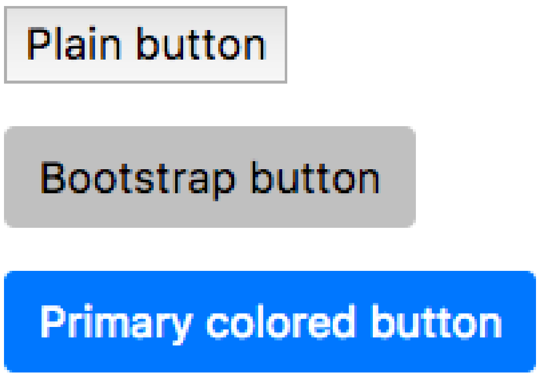


Figure 9.2: Bootstrap Button classes.

The last button in this example has an additional utility class `btn-primary` that specifies which color style it should use. Bootstrap provides a number of pre-defined color styles, named after semantic meaning (e.g., `primary` or `warning`):

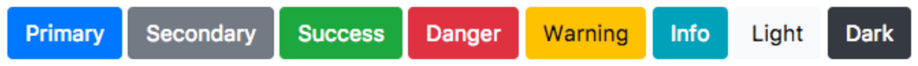


Figure 9.3: Bootstrap Button colors.

These same color classes can be applied to text elements (e.g., `text-primary`), element backgrounds (e.g., `bg-primary`), alert blocks (e.g., `alert-warning`), and more. Modifying the SASS source code for Bootstrap will allow you to easily customize these colors while keeping the theme consistent.

Note that you can even use the `btn` class to style other elements (e.g., `<a>` links) so that they *look* like buttons!

```
<a class="btn btn-primary" href="#">A link</a>
```

"Stacking" utility classes like this is quite common in Bootstrap; it is not unusual for an element to have 4 or 5 different utility classes to fully specify its appearance.

Bootstrap is filled with utility classes, both for generic styling (e.g., borders or position), and for specific types of content (e.g., lists, tables, forms). Check out the documentation and examples for details on how to style particular elements and achieve specific effects.

Bootstrap also includes utility classes specifically to support screen readers. For example, the `sr-only` class can be used to style an element so it is only perceivable to screen readers.

Utility classes do **not** mean you never have to write any CSS rules! While Bootstrap provides classes to handle lots of "common" styling, you'll almost always need to add your own customizations on top of it to produce the appearance you want.

Responsive Design

A major feature of Bootstrap (and other CSS frameworks) is the ability to support **mobile-first, responsive webpages**. Many Bootstrap utility classes use *media queries* to cause them to respond to the device's screen size.

Bootstrap's responsive utilities utilize a predefined set of responsive breakpoints: screen widths that mark the "boundaries" between different size devices.

Screen Width	Abbreviation	Approximate Device Size
< 576px	(default)	Extra-small devices (portrait phones)
576px	`sm`	Small devices (landscape phones)
768px	`md`	Medium devices (tablets)
992px	`lg`	Large devices (desktops)
1200px	`xl`	Extra-large devices (large desktops)

All of these size specifications are "greater than or equal", following a mobile first approach. Thus a responsive utility that applies to a medium (`md`) screen will also apply to large (`lg`) and extra-large (`xl`) screens.

Responsive utilities are named with the size's abbreviation to specify what screen size that utility should apply on. For example, the `float-left` class will cause an element to float to the left on *any* sized screen (it has no size specification). On the other hand, the `float-md-left` class will cause the element to float **only on medium *or larger* screens**. Similarly, the `float-lg-left` class would cause the element to float only on *large or larger* screens. This would allow you to have an image that floats to the side on a bigger screen, but is flowed with the rest of the content when there isn't enough room.

- The breakpoint abbreviations are consistent across Bootstrap (and even shared by other frameworks such as Materialize), and usually comes after the name of property to change but before the value to give it. For example, the `d-none` class gives an element a `display:none` on all devices, while `d-sm-none` would apply on small or larger screens.

Many (but not all!) Bootstrap classes support responsive breakpoints. The most common class to do this is the `container` class. Containers are the most basic layout element in Bootstrap, and are required for doing more complex layouts (such as Grids, below). The `container` class gives the element a fixed width that includes some padding on the sides, so that the content isn't right up against the edge of the window. If the size of the window shrinks to the point where the padding would disappear, the `container` responsively changes size so that there remains padding even on the smaller screen!

- See this Code Pen for an example. If you slowly resize the browser, you'll see the text "jump" to a smaller width as you pass each responsive breakpoint.
- `.container` elements are usually direct children of the `<body>` (e.g., `<header>`, `<main>`). However, if you want a background color or image to "full bleed" to the edges of the viewport, you can put the `.container` inside the another element that is styled with the background, as in the CodePen example.
- Bootstrap also provides an alternate `container-fluid` class that always has the same amount of padding around the content, with the content

"reflowing" based on the browser size.

In general, every page you style with Bootstrap will have a Container (usually a `.container`) to hold the content.

The Grid

After Containers, most common use of the responsive utility classes in CSS frameworks such as Bootstrap is to provide a **grid layout**, similar to the multi-column layouts created with Flexbox in the previous chapters. The Bootstrap Grid system allows you to specify these kinds of complex, responsive layouts without needing to implement your own Flexbox containers and items.

You specify that a set of elements should be part of a "grid" by putting them inside a `.row` element (the `row` class ensures that the elements are lined up). A `.row` acts a lot like a "flex-container" (and in fact does have `display:flex`), where each of its children will be "flex items". Note that the `.row` must be a *direct* child of a Container (`container` or `container-fluid`); this ensures that grid will be positioned and spaced within its parent correctly.

```html
<div class="container"> <!-- container for the grid -->
    <div class="row"> <!-- a row of items -->
        <div class="col-6">Item 1</div>
        <div class="col-4">Item 2</div>
        <div class="col">Item 3</div>
    </div>
</div>
```

You specify how wide each item in the `.row` is by specifying the number of "columns" it spans. Think of the grid as containing **12** "slots":

Each item in the grid is given a class that indicates how many of these slots it takes up (similar to merging cells in a spreadsheet):

These classes are named in the format `col-#`, where the `#` is how many column slots it should take. For example a `.col-4` will take up 4 columns, while a `.col-6` will take up 6 columns. A class of just `col` will take up an *equal share of the remaining columns* (similar to what happens with the `flex-grow` property!)

Thus the above HTML will produce elements with the following layout:

- Notice that "Item 1" takes up half (6/12) of the grid, "Item 2" takes up a third (4/12), and "Item 3" takes up the remainder (2/12).

You can also make this grid layout responsive by using responsive utility versions of the `col-#` classes. For example, a `.col-md-4` would take up 4 columns *only on medium or larger* screens, while a `.col-lg-6` would take up 6 columns *only* *on large or larger* screens. Note that a child of the `.row` that doesn't have a column size will effectively act as its own row (on its own line).

Finally, note that you can specify *multiple responsive utilities* on each grid item! For example, you can specify that items should take up 6 columns on small or larger screens (`col-6-sm`), only but 3 columns on medium or larger screens (`col-3-sm`):

```
<div class="container">
    <div class="row">
        <div class="col-sm-6 col-md-3">Item 1</div>
        <div class="col-sm-6 col-md-3">Item 2</div>
        <div class="col-sm-6 col-md-3">Item 3</div>
        <div class="col-sm-6 col-md-3">Item 4</div>
    </div>
</div>
```

(See a CodePen of this code in action!)

The ordering of Bootstrap's CSS means that the `col-sm-6` class will get overridden by the `col-md-3` class (since `md` is a more specific size than `sm`), so you can specify both classes and the correct one will apply—the order of the class names in the element doesn't matter. Note that the *"or larger"* part of the responsive breakpoint definition also means that you don't need to specify a column width for *each* different breakpoint, only for the smallest level at which the utility should apply (e.g., we don't need `col-lg-` since it isn't different than `col-md-`).

Bootstrap includes a number of utilities that can be used to customize how grids of content are displayed, including all of the features of Flexbox. For more details and options, see the Grid documentation.

Components

In addition to its basic and responsive utilities, Bootstrap provides a number of CSS classes that let you easily create more complex **components** or widgets: alerts, cards, navbars, collapsible boxes, image carousels, pop-up modals, and more. Each of these components is discussed in the documentation, and has its own particular element structure and CSS classes you need to use.

For example, you can create a card (a styled content container) by creating an element with the `card` class and `card-body` child. You can then include other nested elements with classes such as `card-img-top`, `card-title`, or `card-text` to produce further styled effects:

```
<div class="card"> <!-- the card -->
    <img class="card-img-top" src="..." alt="image at top of card image">
    <div class="card-body"> <!-- the "body" of the card -->
```

```
        <h4 class="card-title">A Card Title</h4>
        <p class="card-text">Some card text that appears below the card title.</p>
        <a href="#" class="card-link">Action Link 1</a>
        <a href="#" class="card-link">Action Link 2</a>
    </div>
</div>
```

(See this code in CodePen).

Many of these components include some form of **interactivity**: for example, the navbar will "collapse" into a hamburger menu on small devices, while modals add in-page pop-up dialogs.

CSS is not capable of adding this level of interactivity—that requires **JavaScript**. Thus to support these components, you also need to include the Bootstrap JavaScript library (as well as its dependency).

```
<!-- dependency: jQuery -->
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
<!-- dependency: popper -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.0/umd/popper.min.

<!-- the Bootstrap JavaScript library -->
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.0/js/bootstrap.min.js"
```

See the chapter on JavaScript for more details on including JavaScript libraries. Note that not all components or utilities require JavaScript.

However, with most Bootstrap components, you don't actually need to write any JavaScript code. Instead, you just need to provide some particular `data-` HTML attributes, and the library will take care of everything else.

- A data attributes is an normal HTML attribute that begins with the text `data-` (e.g., `data-thingamabob` or `data-my-attribute`). The HTML specification explicitly says that attributes that begin with `data-` are legal but not part of the standard, thus enabling them to be used for storing app-specific information. They effectively act as custom "variables" for elements that do not violate the HTML standard.

For example, consider the below code for defining a modal and a button to open it:

```
<!-- note that the `id` attribute is set to "exampleModal" -->
<div id="exampleModal" class="modal fade" tabindex="-1" role="dialog" aria-labelledb
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 id="exampleModalLabel" class="modal-title">Modal title</h5>
        <button type="button" class="close" data-dismiss="modal" aria-label="Close">
```

```html
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="modal-body">Some content shown in the body of the modal</div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary">Save changes</button>
      </div>
    </div>
  </div>
</div>

<!-- data-toggle must be set to "modal" and data-target must be set to
the value of the `id` attribute on the element that starts your modal markup -->
<button type="button" class="btn btn-primary" data-toggle="modal" data-target="#exampleModal">
```

(Try it out in CodePen).

There is a lot to unpack in this code, including styling classes and ARIA attributes (e.g., `role`, `aria-labelledby`, and `aria-hidden`). The important part is that the root `.modal` element is *hidden* (`display:none`) when the page first loads. But when you click on the `<button>`, the included JavaScript library sees that it has a `data-toggle="modal"` attribute, and determines that clicking the button should show the modal. The JavaScript using the `data-target` attribute on that button to determine *which* modal to open (finding the modal with that `id`), and then shows it. The `data-dismiss` attribute on the first `<button>` inside the modal allows it to be closed.

The best way to understand these components is to start from the examples in the documentation, and then consider each element and attribute one at a time. Read the documentation!

The JavaScript included in these CSS frameworks allows you to easily add interactive features, but **they do not mix well with more advanced JavaScript frameworks like React.** To work with React, you'd need to switch instead to a library such as reactstrap, which uses the Bootstrap 4 CSS stylesheet, but re-implement the JavaScript portions to fit into the React framework.

Resources

- Bootstrap 4 Documentation
- Materialize Documentation
- Foundation Documentation
- Material Components for the Web Documentation
- `normalize.css` Documentation

- Bootstrap 3 Tutorial (w3schools) For an older version of bootstrap, but many of the componets and examples are similar.

This chapter was borrowed to a large extent from a tutorial by David Stearns.

Chapter 10

JavaScript Fundamentals

HTML and CSS are *markup languages*, used to annotate websites to describe their meaning and appearance. But they are not "full" (i.e., Turing Complete) programming languages—they don't have variables, control structures, or other features required for the computer to execute an *algorithm*. So to make *interactive* websites or complex web applications, you need a complete programming language. And the language used by browsers is **JavaScript**. This chapter introduces the fundamentals of the JavaScript language as used in web development, focusing on variables, data types, and basic control structures. Functions are introduced, but addressed in more detail in the next chapter.

Note, this course assumes you already are already familiar with introductory programming in a imperative language such as Java. It focuses on how JavaScript differs from languages such as Java, rather than foundational programming concepts. If you need review on those topics, you can check out a basic programming tutorial or review your notes from a previous programming course.

10.1 Programming with JavaScript

JavaScript is a **high-level, general-purpose programming language**, allowing you to declare instructions for the computer in an almost-human readable way (similar to Java). JavaScript is an *imperative* language, so you write algorithms as step-by-step instructions (lines of code) using JavaScript's syntax, and the computer will interpret these instructions in order to execute the algorithm. Browsers are able to download scripts written in JavaScript, executing them line-by-line and using those instructions to manipulate what content is displayed.

Indeed, JavaScript is an **interpreted language**, in that the computer (specifically a *JavaScript Interpreter*) translates the high-level language into machine language *on the fly at runtime.* The interpreter will read and execute one line of code at a time, executing that line before it even begins to consider the next. This is in contrast with *compiled languages* like C or Java that have the computer do the translation in one pass (at compile-time), and then only execute the program after the whole thing is converted to machine language.

- This on-the-fly translation means that interpreted languages like JavaScript are usually slower than compiled languages, but not enough that we'll notice (web browsers include *highly* optimized interpreters). The bigger drawback is that without the compile step, program errors appear at runtime rather than compile time, making them more difficult to catch and fix. As such, JavaScript developers make heavy use of various linting tools (which flag common problems that can lead to runtime errors), as well as automated testing systems to check against a variety of inputs.

  Many JavaScript editors do this kind of error checking—for example, VS Code provides syntax error checking out of the box, with more detailed linting available through extensions. See this article for more code features that support programming in JavaScript, including IntelliSense and type checking.

Although JavaScript was designed for and is most commonly used within web browsers (which contain their own JavaScript Interpreters), it can also be executed on the command line by using Node.js, allowing JavaScript to be a fully general language. Both techniques are described in this chapter.

History and Versions

The JavaScript language was developed by Brendan Eich (the co-founder of Mozilla) in 1995 while he was working for Netscape. The original prototype of the language was created in 10 days… a fact which may help explain some of the quirks in the language.

- Despite the names, *JavaScript* and the *Java* language have nothing to do with one another (and are in fact totally separate programming languages used in drastically different contexts). JavaScript was named after Java as a marketing ploy to cash in on the popularity of the latter. It is the programming equivalent of the Transmorphers movie.
- The JavaScript language is officially an implementation of ECMAScript (named for the European Computer Manufacturers Association) specification. Hence versions of JavaScript are labeled with the prefix "ES". For example, this chapter describes `ES5` (JavaScript 5) syntax.

Like HTML and CSS, the JavaScript language continues to be developed and refined through new versions. Each new version of JavaScript includes additional syntax shorts, specialized keywords, and additional core functions, but otherwise are compatible with previous code. The main limitation on utilizing new JavaScript features is whether the *interpreters* found in web browsers are able to support them.

This course primarily utilizes syntax and features for `ES5`, which was introduced in 2009 and today is supported by all modern browsers (i.e., IE 10 or later). Later chapters will describe features of `ES6` (also called `ES2015`), which was introduced in 2015 and works on many browsers (e.g., Microsoft Edge but not IE 11). `ES7` was finalized in 2016 but is still not reliably supported.

- Note that Microsoft no longer supports earlier versions of IE, but a small percentage of desktops still run Windows XP with IE 8. If you must support these browsers, you can often use a "transpiler" like Babel to translate modern JavaScript source code into an earlier version.

10.2 Running JavaScript

There are two primary ways of executing code written in JavaScript:

In the Browser

JavaScript scripts are most commonly executed in a web browser as part of the browser rendering a web page. Since browsers render HTML content (in `.html` files), JavaScript scripts are included in that HTML by using a `<script>` tag and specifying the *relative* path to the a file containing the code (usually a **`.js`** file) to execute. When the browser renders the HTML (reading top to bottom) and gets to that tag, it will download and execute the specified script file using the JavaScript interpreter:

```html
<!-- execute the script.js file -->
<script src="path/to/my/script.js"></script>
```

- Notice that the `<script>` element is *not* empty! It is possible to include JavaScript code directly inside the tag, but this is considered bad practice (keep concerns separated!) and should only be used for quick tests.

The `<script>` tag can be included anywhere in an HTML page. Most commonly it is either placed in the `<head>` in order for the script to be executed *before* the page content loads, or at the very end of the `<body>` in order for the script to be executed *after* the page content loads (and thus allows the JavaScript to immediately interact with the loaded HTML elements). We will (begin by) most commonly be putting `<script>` tags at the end of the `<body>`:

```html
<!DOCTYPE html>
<html>
<head>
```

```
  <!-- include here to run before the page appears -->
  <script src="js/script.js"></script>
</head>
<body>
   ... content ...

   <!-- include here to run "after" html appears -->
   <!-- we will usually do this -->
   <script src="js/script.js"></script>
</body>
<html>
```

- Most browsers also support adding a `defer` attribute to the `<script>` element that tells the browser to defer downloading and running the script until the page is fully loaded and rendered to the screen. With this attribute, you can put the `<script>` element in the `<head>` without any issues. However, this attribute is not supported in IE 9 or earlier; it is thus still best practice to put the `<script>` element at the end of the body.

While JavaScript most commonly is used to manipulate the web page content and is thus pretty obvious to the user, it *also* can produce "terminal-like" output—including printed text and **error messages**. This output can be viewed through the **JavaScript Console** included as a *developer tool* in the Chrome browser (other browsers include similar tools):

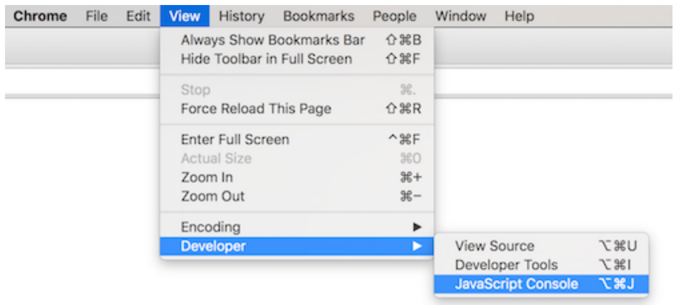

Chrome File Edit View History Bookmarks People Window Help
Always Show Bookmarks Bar ⇧⌘B
Hide Toolbar in Full Screen ⇧⌘F
Stop ⌘.
Force Reload This Page ⇧⌘R
Enter Full Screen ^⌘F
Actual Size ⌘0
Zoom In ⌘+
Zoom Out ⌘−
Encoding
Developer
View Source ⌥⌘U
Developer Tools ⌥⌘I
JavaScript Console ⌥⌘J

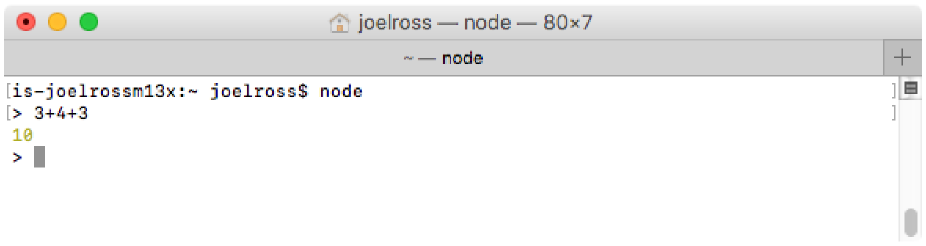
Figure 10.1: Accessing the developer console in Chrome.

Important: You should ***always*** have the JavaScript Console open when developing JavaScript code, since this is where any error messages will appear!

On the Command-line with Node.js

It is also possible to execute JavaScript code on the command line using Node.js. Node is a **command line runtime environment**—that is, a JavaScript interpreter that can be run on the command line.

With Node installed on your machine, you can use the **node** terminal command to start an *interactive Node session*. This will allow you to type JavaScript code directly in the terminal, and your computer will interpret and execute each line of code:



```
is-joelrossm13x:~ joelross$ node
> 3+4+3
10
> 
```

Figure 10.2: An interactive Node session.

You can enter one line of code at a time at the prompt (`>`). This is a nice way to experiment with the JavaScript language or to quickly run and test some code.

- You can exit this session by typing the `quit()` command, or hitting `ctrl-z` (followed by Enter on Windows).
- Note that the JavaScript console in the Chrome Developer tools provides the same interactive functionality.

It is also possible to run entire scripts (`.js` files) from the command line by using the `Node` command and specifying the script file you wish to execute:

```
node my-script.js
```

This allows you author entire programs in JavaScript (e.g., writing the code using VS Code), and then executing them from the command line. This process is more common when doing *server-side* web development (such as implementing a web server using Node), but we will still use it for practice and testing.

10.3 Writing Scripts

Unlike Java or C, JavaScript has no `main()` method that "starts" the program.

Instead, a JavaScript script (a `.js` file) contains a sequence of **statements** (instructions) that the interpreter executes *in order, one at a time, top to bottom*—just as if you had typed them into an interactive session one after another. Each line can declare a variable or function, which will then be available to the next statement to use or call. Thus with a JavaScript program, you should think of the *sequence* of steps you want to be performed, and write lines of code in that order.

- In a way, you can think of the entire file as the "body" of a `main()` method... except that this body will contain further function declarations.

For example:

```
/* script.js */
/* This is the ENTIRE contents of the file! */
console.log("Hello world!");  //this is executed first
console.log("I'm doing JavaScript!");  //this is executed second
```

The above example code demonstrates the **`console.log()`** function, which is JavaScript's equivalent to Java's `System.out.println()`—the output will be shown in the JavaScript console (in the Developer Tools). Thus we talk about "logging out" values in JavaScript, instead of "printing" values in Java.

- The `console.log()` function is technically a `log()` method belonging being called on a *global* `console` object. Globals and objects will be discussed in more detail below.

Figure 10.3: Don't forget the semicolons!

As in Java, each statement should end with a semicolon (**;**) marking the end of that statement. But unlike Java, JavaScript can be forgiving if you forget a semicolon. The JavaScript interpreter tries to be "helpful" and will often assume that statements end at the end of a line if the next line "looks like" a new statement. However, it occasionally screws up in horrible ways—and so best practice as a developer is to **always include the semicolons**.

Strict Mode

`ES5` includes the ability for JavaScript to be interpreted in **strict mode**. Strict mode is more "strict" about how the interpreter understands the syntax: it is less likely to assume that certain programmer mistakes were intentional (and so try to run the code anyway). For example, in *strict mode* the interpreter will produce an *Error* if you try and use a variable that has not yet been defined, while without strict mode the code will just use an `undefined` value. Thus working in strict mode can help catch a lot of silly mistakes.

You declare that a script or function should be executed in strict mode by putting an *interpreter declaration* at the top:

```
'use strict';
```

- This is not a String (or even JavaScript code!), but rather a *declaration* to the interpreter about how that interpreter should behave.

ALWAYS USE STRICT MODE! It will help avoid typo-based bugs, as well as enable your code to run more efficiently.

10.4 Variables

Variables in JavaScript are **dynamically typed**. This means that variables are not *not* declared as a particular type (e.g., `int` or `String`), but instead take on the data type (`Number`, `String`, etc.) of the *value* currently assigned to that variable. As we don't specify the data type, JavaScript variables are declared using the `let` keyword:

```
let message = 'Hello World';  //a String
console.log(typeof message);  //=> `string`

let shoeSize = 7;  //a Number
console.log(typeof shoeSize);  //=> 'number'
```

- The `typeof` operator will return the data type of a variable. It is not widely used outside of debugging.
- As in Java, JavaScript variables should be given descriptive names using camelCase.

  Pro Tip: Even though variables in JavaScript loosely typed, the data type of a value is still important! In order to help keep track of the type of each variable in JavaScript, include the type in the variable name. For example: `textString`, `wordsArray`, `totalNum`, `itemStr`, etc..

Declared variables have a default value of `undefined`—a value representing that the variable has no value. This is somewhat similar to `null` in Java (though

JavaScript *also* as an `null` value that is not commonly used):

```
//create a variable (not assigned)
let hoursSlept;
console.log(hoursSlept);  //=> undefined
```

Note that the `let` keyword is in fact new syntax introduced with `ES6`: older versions of JavaScript used the `var` keyword instead (and you will see `var` in most existing examples and tutorials). The difference is that variables declared with `let` are **"block scoped"**, meaning they are only available within the *block* (the `{}`) in which they are defined. This is the same way variables are scoped in Java. Variables declared with `var`, on the other hand, are "functionally scoped" so are available anywhere within the *function* in which they are defined. This means that you could declare a variable within an `if` block, and that variable would continue to be available outside that block! Thus `let` allows for cleaner, more efficient code, and with less bugs.

- `let` is the one `ES6` feature supported by IE 11, meaning it can be used with most current browsers. However, if you do need to support an older browser (e.g., IE 10, Safari 9.3, Android 4.4), you should transpile your code or stick to using `var`

Along with `let`, JavaScript variables can also be declared using the `const` keyword to indicate that they are *constant* (similar to what the `final` keyword does in Java). A `const` variable is block scoped, but can only be assigned once:

```
const ISCHOOL_URL = 'https://ischool.uw.edu'; //declare constant
ISCHOOL_URL = 'https://example.com'; //TypeError: Assignment to constant variable.
```

Basic Data Types

JavaScript supports many of the same basic data types as Java and other languages:

- **Numbers** are used to represent numeric data (JavaScript does not distinguish between integers and floats). Numbers support the same *mathematical* and operators as Java. Common mathematical functions can be accessed through in the built-in `Math` global (similar to Java's `Math` class).

```
let x = 5;
typeof x;  //'number'
let y = x/4;
typeof y;  //'number'

//numbers use floating point division
console.log( x/4 );  //1.25

//use the Math.floor() function to do integer division
```

```
console.log( Math.floor(x/4) );  //1

//other common Math functions available as well
console.log( Math.sqrt(x) );  //2.23606797749979
```

- **Strings** can be written in either single quotes (`'`) or double quotes (`"`), but most style guidelines recommend single quotes—just be consistent! Strings can be concatenated as in Java:

  ```
  let name = 'Joel';
  let greeting = 'Hello, my name is '+name; //concatenation
  ```

  Strings also support many methods for working with them. As with Java, JavaScript strings are *immutable*, so most string methods return a new, altered string.

  ```
  let message = 'Hello World';
  let shouted = message.toUpperCase();
  console.log(shouted);  //=> 'HELLO WORLD'
  ```

- **Booleans** (`true` and `false`) in JavaScript work the same way as in Java, can be produced using the same *relational operators* (e.g., `<`, `>=`, `!=`), and support the same *logical operators* (e.g., `&&`, `||`, and `!`):

  ```
  //conjunction (and)
  boolOne && boolTwo

  //disjunction (or)
  boolOne || boolTwo

  //negation (not)
  !boolOne //not
  ```

  See Type Coercion and Control Structures below for more on working with Booleans in JavaScript.

Arrays

JavaScript also supports **arrays**, which are *ordered, one-dimensional sequences of values*. As in Java, JavaScript Arrays are written as literals inside square brackets `[]`. Individual elements can be accessed by (0-based) *index* using **bracket notation**.

```
//an array of names
let names = ['John', 'Paul', 'George', 'Ringo'];

//an array of numbers (can contain "duplicate" values)
```

```
let numbers = [1, 2, 2, 3, 5, 8];

//arrays can contain different types (including other arrays!)
let things = ['raindrops', 2.5, true, [3, 4, 3]];

//arrays can be empty (contain no elements)
let empty = [];

//access using bracket notation
console.log( names[1] );  // "Paul"
console.log( things[3][2] );  // 3

numbers[0] = '343';  //assign new value at index 0
console.log( numbers );  // [343, 2, 2, 3, 5, 8]
```

Note that it is possible to assign a value to *any* index in the array, even if that index is "out of bounds". This will *grow* the array (increase its length) to include that index—intermediate indices will be given values of `undefined`. The *length* of the array (accessed via the `.length` attribute) will always be the index of the "last" element + 1, even if there are fewer defined values within the array.

```
let letters = ['a', 'b', 'c'];
console.log(letters.length);  // 3
letters[5] = 'f';  //grows the array
console.log(letters);  // [ 'a', 'b', 'c', , , 'f' ]
                       //blank spaces are undefined
console.log(letters.length);  // 6
```

Arrays also support a variety of methods that can be used to easily modify their elements, similar to the `ArrayList` class in Java:

```
//Make a new array
let array = ['i','n','f','o'];

//add item to end of the array
array.push('343');
console.log(array); //=> ['i','n','f','o','343']

//combine elements into a string
let str = array.join('-');
console.log(str); //=> "i-n-f-o-343"

//get index of an element (first occurrence)
let oIndex = array.indexOf('o'); //=> 3

//remove 1 element starting at oIndex
array.splice(oIndex, 1);
```

```
console.log(array); //=> ['i','n','f','343']
```

Objects

The most generic and useful data type in JavaScript is the Object data type. An **Object** is a lot like an array in that it is in that it is a (one-dimensional) sequence of values that are all stored in a single variable. However, rather than using *integers* as the index for each element, an Object uses *Strings*. Thus Objects are *unordered* sequences of **key-value pairs**, where the keys (called **"properties"**) are arbitrary Strings and the values are any data type—each property can be used to *look up* (reference) the value associated with it.

- This is a lot like a real-world dictionary or encyclopedia, in which the words (keys) are used to look up the definitions (values). A phone book works the same way (the names are the keys, the phone numbers are the values). A JavaScript Object provides a *mapping* from properties to values.

- JavaScript Objects are similar data structures to Java *HashMaps*, Python *dictionaries*, R *lists*, or even an HTML `<dl>` descriptive list! More generally, these are known as *associative arrays* (they are arrays that "associate" a key and a value). As with other associative arrays, JavaScript Objects are implemented as hash tables, making data access very fast.

Objects are written as literals inside curly braces `{}`. Property-value pairs are written with a *colon* (`:`) between the property name and the value, and each element (pair) in the Object is separated by a *comma* (`,`). Note that the property names do *not* need to be written in quotes if they are a single word (the quotes are implied—properties are always Strings):

```
//an object of ages (explicit Strings for keys)
//The `ages` object has a `sarah` property (with a value of 42)
let ages = {'sarah':42, 'amit':35, 'zhang':13};

//different properties can have the same values
//property names with non-letter characters must be in quotes
let meals = {breakfast:'coffee', lunch: 'coffee', 'afternoon tea': 'coffee'}

//values can be of different types (including arrays or other objects!)
let typeExamples = {number:12, string:'dog', array:[1,2,3]};

//objects can be empty (contains no properties)
let empty = {}
```

Important note: Objects are an **unordered** collection of key-value pairs! Because you reference a value by its *property name* and not by its position (as you do in an array), the exact ordering of those elements doesn't matter—the interpreter just goes immediately to the value associated with the property. This almost means that when you log out an Object, the order in which the properties are printed may not match the order in which you specified them in the literal.

- If you `console.log()` an object, you will usually see a nicely format version of that object. However, if you try to convert that object to a *string* (e.g., via concatenation: `let output = "my object: "+myObject`), you will instead be presented with the "string version" of that object: `[object Object]`. This is not an array, but the String version of an object (similar to the hash you get when you try and print a Java array directly). You can instead use the fact that `console.log` accepts *multiple parameters* to output an object with a leading string:

  ```
  let myObject = {a:1, b:2}

  //convert object to string, won't log nicely
  console.log("My object: " + myObject); //=> My object: [object Object]

  //log the object directly
  console.log("My object ", myObject); //=> My object {a: 1, b: 2}
  ```

Despite the name, JavaScript Objects have nothing to do with Object-Oriented Programing. However, they can be used to represent "things" in the same way as a Java object, with each property acting like an "attribute" (instance variable):

```
//an object representing a Person (spacing is for readability; white-space is ignore
let person = {
  firstName: 'Alice',
  lastName: 'Smith',
  age: 40,
  pets: ['rover', 'fluffy', 'mittens'],  //value is an array
  favorites: {   //value is another object
    music: 'jazz',
    food: 'pizza',
    numbers: [12, 42]  //value is an array
  }
}
```

Accessing Objects

Similar to arrays, Object values can be access via **bracket notation**, specifying the *property name* as the index. If an object does not explicitly have a property value, accessing that key produces `undefined` (the property's value is `undefined`).

```
let favorites = {music: 'jazz', food: 'pizza', numbers: [12, 42]};

//access variable
console.log( favorites['music'] ); //'jazz'

//assign variable
favorites['food'] = 'cake';  //property name is a string
console.log( favorites['food'] ); //'cake'

//access undefined key
console.log( favorites['language'] ); //undefined
favorites['language'] = 'javascript'; //assign new key and value

//access nested values
console.log( favorites['numbers'][0] ); //12

//use a variable as the "key"
let userInputtedTopic = 'food'; //pretend this value is supplied dynamically
console.log(favorites[userInputtedTopic]);  //'cake'
```

Additionally, Object values can also be accessed via **dot notation**, as if the properties were *public attributes* of a class instance. This is often simpler to write and to read: remember to read the `.` as an `'s`!

```
let favorites = {music: 'jazz', food: 'pizza', numbers: [12, 42]};

//access variable
console.log( favorites.music ); //'jazz'

//assign variable
favorites.food = 'cake';
console.log( favorites.food ); //'cake'

//access undefined key
console.log( favorites.language ); //undefined
favorites.language = 'javascript'; //assign new key and value

//access nested values
console.log( favorites.numbers[0] ); //12
```

- The only advantage to using *bracket notation* is that you can specify property names as variables or the results of an expression. But overall, the recommendation is to use *dot notation* unless the property you wish to access is dynamically determined.

It is possible to get an *array* of an object's keys calling the `Object.keys()` method and passing in the object you wish to get the keys of. Note that an equivalent function for values is not supported by most browsers; a better approach is to iterate through the keys to identify all the values.

```
let ages = {sarah:42, amit:35, zhang:13};
let keys = Object.keys(ages);  // [ 'sarah', 'amit', 'zhang' ]
```

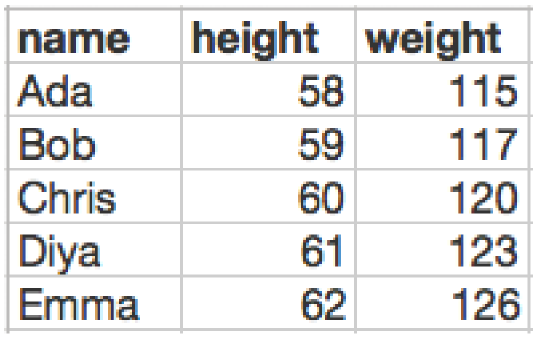
Arrays of Objects

As noted above, both arrays and objects can have values of any type—including other arrays or objects! The ability to nest objects inside of objects is incredibly powerful, and allows us to define arbitrarily complex information structurings (schemas). Indeed, most data in computer programs—as well as public information available on the web—is structured as a set of nested maps like this (though possibly with some level of abstraction).

One of the most common forms of nesting you'll see is to have an **array of objects** where each object has *the same properties* (but different values). For example:

```
//an arbitrary list of people's names, heights, and weights
let people = [
    {name: 'Ada', height: 58, 'weight': 115},
    {name: 'Bob', height: 59, 'weight': 117},
    {name: 'Chris', height: 60, 'weight': 120},
    {name: 'Diya', height: 61, 'weight': 123},
    {name: 'Emma', height: 62, 'weight': 126}
]
```

This structure can be seen as a list of **records** (the objects), each of which have a number of different **features** (the key-value pairs). This list of feature records is in fact a common way of understanding a **data table** like you would create as an Excel spreadsheet:



name	height	weight
Ada	58	115
Bob	59	117
Chris	60	120
Diya	61	123
Emma	62	126

Figure 10.4: A data table.

Each object (record) acts as a "row" in the table, and each property (feature) acts as a "column". As long as all of the objects share the same keys, this array of objects *is* a table!

Type Coercion

As mentioned above, variables in JavaScript are *dynamically typed*, and thus have a data type of the value **currently** assigned to that variable. JavaScript variables are able to "change type" by having a different type of data assignment to them:

```
let myVariable = 'hello'; //value is a String
myVariable = 42; //value is now a Number
```

Unlike some other dynamically typed variables, JavaScript will not throw errors if you try to apply operators (such as `+` or `<`) to different types. Instead, the interpreter will try to be "helpful" and **coerce** (convert) a value from one data type into another. While this process is similar to how Java will automatically cast data types into Strings (e.g., `"hello"+4`), JavaScript's type coercion can produce a few quirks:

```
let x = '40' + 2;
console.log(x);  //=> '402'; the 2 is coerced to a String
let y = '40' - 4;
console.log(y);  //=> 36; can't subtract strings so '40' is coerced to a Number!
```

JavaScript will also attempt to coerce values when checking for equality with `==`:

```
let num = 10
let str = '10'

console.log(num == str) //true, the values can be coerced into one another
```

In this case, the interpreter will coerce the Number `10` into the String `'10'` (since numbers can always be made into Strings), and since those Strings are the same, determines that the variables are equal.

In general this type of automatic coercion can lead to subtle bugs. Thus you should instead always use the `===` operator (and it's partner `!==`), which checks both value *and* type for equality:

```
let num = 10
let str = '10'

console.log(num === str) //false, the values have different types
```

JavaScript will do its best to coerce any value when compared. Often this means converting values to Strings, but it will also commonly convert values into *booleans* to compare them. So for example:

```
//compare an empty String to the number 0
console.log( '' == 0 ); //true; both can be coerced to a `false` value
```

This is because both the empty string `''` and `0` are considered **"falsey"** values (values that can be coerced to `false`). Other falsy values include `undefined`, `null`, and `NaN` (not a number). All other values will be coerced to `true`.

For more examples of the horror of JavaScript coercion, see this video (about 1:20 in).

10.5 Control Structures

JavaScript control structures have a similar syntax to those in Java or C. For example, a JavaScript **`if` statement** is written as:

```
if(condition){
  //statements
}
else if(alternativeCondition) {
  //statements
}
else {
  //statements
}
```

The **`condition`** can be any expression that evaluates to a Boolean value. But since any value can be *coerced* into Booleans, you can put any value you want inside the `if` condition. This is actually really useful—since `undefined` is a falsy value, you can use this coercion to check if a variable has been assigned or not:

```
//check if a `person` variable has a `name` property
if(person.name){
    console.log('Person does have a name!');
}
```

In the above example, the condition will only coerce to `true` if `person.name` is defined (*not* `undefined`) and is not empty. If somehow the variable has not been assigned (e.g., the user didn't fill out the form completely), then the condition will not be true.

- While using `person.name !== undefined` is an equivalent expression, it is more idiomatic to utilize the coercion and have plain variable as the conditional expression.

Additionally, JavaScript provides a ternary conditional operator that lets you write a simple `if` statement as a single expression:

```
let x; //declare variable
if(condition) {
    x = 'foo';
} else {
    x = 'bar';
}

//can be condensed into:
let x = condition ? 'foo' : 'bar';
```

- This expression is read as "*if* `condition` *then* resolve to `'foo'` *else* resolve to `bar`". Each part (the condition, the "if true" result, and the "if false" result) can be any expression, though you should keep them simple to avoid confusion.

JavaScript also supports **`while`** loop (for indefinite iteration) and **`for`** loops (for definite iteration) similar to Java. The only difference is that because JavaScript variables are dynamically typed, the *loop control variables* are not declared with a type:

```
//an example for loop. The `i` is not declared as an int
for(let i=0; i<array.length; i++){
  console.log(array[i]);
}
```

JavaScript *does* have a `for ... in` syntax. However, it doesn't work as you would expect for arrays (it iterates over "enumerable properties" rather than the specific indices), and so should **not** be used with arrays. `ES6` also introduces a `for ... of` syntax for iterating over arrays, but this is not supported by all browsers and so is not recommended. Instead, the current best practice is to use the above `for` loop, or better yet the `forEach()` method described in the next chapter.

- If you need to iterate over the keys of an object, use the `Object.keys()` method to get an array to loop through!

10.6 Functions

And of course, JavaScript includes **functions** (named sequences of statements used to *abstract* code). JavaScript functions are written using the following syntax:

```
//A function named `makeFullName` that takes two arguments
//and returns the "full name" made from them
```

```
function makeFullName(firstName, lastName) {
  //Function body: perform tasks in here
  let fullName = firsName + " " + lastName;

  // Return: what you want the function to output
  return fullName;
}

// Call the makeFullName function with the values "Alice" and "Kim"
// Assign the result to `myName`
let myName = makeFullName("Alice", "Kim")  // "Alice Kim"
```

- Functions are defined by using the **`function`** keyword (placed before the name of the function) instead of Java's `public static`, don't declare a return type (since the language is dynamically typed), and don't indicate types for the parameters. Otherwise, JavaScript functions have identical syntax to Java functions.

- If a function lacks a `return` value, then that function returns the value `undefined`.

As in other languages, function arguments in JavaScript are implicitly declared *local* variables. However, in JavaScript **all arguments are optional**. Any argument that is *not* passed a specific value will be `undefined`. Any passed in value that does not have a variable declared for its position will not be assigned to a variable.

```
function sayHello(name) {
    return "Hello, "+name;
}

//expected: argument is assigned a value
sayHello("Joel");  //"Hello, Joel"

//argument not assigned a value (left undefined)
sayHello();  //"Hello, undefined"

//extra arguments (values) are not assigned to variables, so are ignored
sayHello("Joel", "y'all");  //"Hello, Joel"
```

- If a function has an argument, that doesn't mean it got a value. If a function lacks an argument, that doesn't mean it wasn't given a value!

In addition to this basic structure, JavaScript functions are frequently used for *functional programming*, as described in the next chapter.

Resources

As the language used for web programming, JavaScript may have more freely available online learning resources than any other language! Some of my favorites include:

- A Re-Introduction to JavaScript a focused tutorial on the primary language features
- You Don't Know JS a free textbook covering all aspects of the JavaScript language. Very readable and thorough, with lots of good examples.
- JavaScript for Cats a gentler introduction for "Scaredy-Cats"
- MDN JavaScript Reference a complete documentation of JavaScript, including tutorials
- w3Schools JavaScript Reference a slightly more friendly reference for the language
- Google's JavaScript Style Guide

Chapter 11

Functional Programming in JS

Despite it's name, the JavaScript language was based more on Scheme than it was on Java. Scheme is a **functional programming language**, which means it follows a programming paradigm centered on *functions* rather than on *variables*, objects, and statements as you've done before (known as *imperative programming*). An alternative to object-oriented programming, functional programming provides a framework for thinking about how to give instructions to a computer. While JavaScript is not a fully functional language, it does support a number of functional programming features that are vital to developing effective and interactive systems. This chapter introduces these functional concepts.

11.1 Functions ARE Variables

Normally you've considered functions as "named sequences of instructions", or groupings of lines of code that are given a name. But in a functional programming paradigm, functions are first-class citizens—that is, they are "things" (values) that can be organized and manipulated *just like variables*.

In JavaScript, **functions ARE variables**:

```
//create a function called `sayHello`
function sayHello(name) {
    console.log("Hello, "+name);
}

//what kind of thing is `sayHello` ?
console.log(typeof sayHello);  //=> 'function'
```

Just like `let x = 3` defines a variable for a value of type `number`, or `let msg = "hello"` defines a variable for a value of type `string`, the above `sayHello` function is actually a variable for a *value* of type `function`!

Important: we refer to a function by it's name *without* the parentheses!

The fact that functions **are** variables is the core realization to make when programming in a functional style. You need to be able to think about functions as **things** (nouns), rather than as **behaviors** (verbs). If you imagine that functions are "recipes", then you need to think about them as *pages from the cookbook* (that can be bound together or handed to a friend), rather than just the sequence of actions that they tell you to perform.

And because functions are just another type of variable, they can be used **anywhere** that a "regular" variable can be used. For example, functions are values, so they can be assigned to other variables!

```
//create a function called `sayHello`
function sayHello(name) {
    console.log("Hello, "+name);
}

//assign the `sayHello` value to a new variable `greet`
let greet = sayHello;

//call the function assigned to the `greet` variable
greet("world");  //logs "Hello world"
```

- It helps to think of functions as just a special kind of array. Just as *arrays* have a special syntax `[]` (bracket notation) that can be used to "get" a value from the list, *functions* have a special syntax `()` (parentheses) that can be used to "run" the function.

Anonymous Functions

Functions are values, just like arrays and objects. And just as arrays and objects can be written as literals which can be *anonymously* passed into functions, JavaScript supports **anonymous functions**:

```
var array = [1,2,3]; //named variable (not anonymous)
console.log(array); //pass in named var
console.log( [4,5,6] ); //pass in anonymous value
```

```
//named function (normal)
function sayHello(person){
    console.log("Hello, "+person);
}
```

```
//an anonymous function (with no name!)
//(We can't reference this without a name, so writing an anonymous function is
//not a valid statement)
function(person) {
    console.log("Hello, "+person);
}

//anonymous function (value) assigned to variable
//equivalent to the version in the previous example
let sayHello = function(person) {
    console.log("Hello, "+person);
}
```

- You can think of this structure as equivalent to declaring and assigning an array `let myVar = [1,2,3]`… just in this case instead of taking the anonymous array (right-hand side) and giving it a name, we're taking an *anonymous function* and giving it a name!

Thus you can define named functions in one of to ways: either by making an explicitly named function or by assigning an anonymous function to a variable:

```
//these produce the same function
function foo(bar) {}
let foo = function(bar) {}
```

The only difference between these two constructions is one of *ordering*. When the JavaScript interpreter is initially reading and parsing the script file, it will put variable and function declarations into memory before it executes any of the file. In effect, JavaScript will seem to "move" variable and function declarations to the top of the file! This process is called **hoisting** (declarations are "hoisted" to the top of the script). Hoisting *only* works for named function declarations: assigning an anonymous function to a variable will not hoist that function's definition:

In practice, you should **always** declare and define functions before you use them (put them all at the top of the file!), which will reduce the impact of hoisting and allow you to use either construction.

11.2 Object Functions

Moreover, functions are values, so they can be assigned as values of *object properties* (since object properties are like name-spaced variables):

```
//an object representing a dog
let dog = {
    name: 'Sparky'
    breed: 'mutt'
}

//assign an anonymous function to the `bark` property
dog.bark = function(){
    console.log('woof!');
}

//call the function
dog.bark(); //logs "woof!"
```

- Again, this is just like how you can assign an array as an object's property. With an array value you would use bracket notation to use it's "special power"; with a function value you use parentheses!

This is how we can create an equivalent of "member functions" (or methods) for individual objects: the `dog` object now has a function `bark()`!

- Similar to Java, you can refer to the *object on which a function is called* using the keyword `this`. Note that the manner in which the `this` variable is assigned can lead to some subtle errors when using callback functions (below). For more details, see the chapter on ES6 features. As a brief example:

```
// An object representing a Dog
let fido = {
  name: "Fido",
  bark: function() { console.log(this.name, "woofs")}
}

// An object representing another Dog
let spot = {
  name: "Spot",
  bark: function() { console.log(this.name, "yips")}
}

console.log('***This is Fido barking:***');
fido.bark(); //=> "Fido woofs". Note, `this` will refer to the `fido` object.

console.log('***This is Spot barking***');
spot.bark()); //=> "Spot yips". Note, `this` will refer to the `fido` object.
```

11.3 Callback Functions

Finally, functions are values, so they can be *passed as parameters to other functions*!

```
//create a function `sayHello`
function sayHello(name){
    console.log("Hello, "+name);
}

//a function that takes ANOTHER FUNCTION as an argument
//this function will call the argument function, passing it "world"
function doWithWorld(funcToCall){
    //call the given function with an argument of "world"
    funcToCall("world");
}

doWithWorld(sayHello);  //logs "Hello world";
```

In this case, the `doWithWorld` function will *execute* whatever function it is given, passing in a value of `"world"`.

- **Important note**: when we pass `sayHello` as an argument, we don't put any parentheses after it! Putting the parentheses after the function name *executes* the function, causing it to perform the lines of code it defines. This will cause the expression containing the function to *resolve* to its returned value, rather than being the function value itself. It's like passing in the baked cake rather than the recipe page.

  ```
  function greet() {  //version with no args for clarity
      return "Hello";
  }

  //log out the function value itself
  console.log(greet);  //logs e.g., [Function: greet], the function

  console.log(greet());  //logs "Hello", which is what `sayHello()` resolves to
  ```

A function that is passed into another is commonly referred to as a **callback function**: it is an argument that the other function will "call back to" and execute when needed.

```
function doLater(callback) {
    console.log("I'm waiting a bit...");
    console.log("Okay, time to work!");
    callback();  //"call back" and execute that function
}
```

```
function doHomework() {
    // ...
};

doLater(doHomework);
```

Functions can take more than one callback function as arguments, which can be a useful way of *composing* behaviors.

```
function doTogether(firstCallback, secondCallback){
    firstCallback();  //execute the first function
    secondCallback();  //execute the second function
    console.log('at the same time!');
}

function patHead() {
    console.log('pat your head');
}

function rubBelly() {
    console.log('rub your belly');
}

//pass in the callbacks to do them together
doTogether(patHead, rubBelly);
```

This idea of *passing functions are arguments to other functions* is at the heart of functional programming, and is what gives it expressive power: we can define program behavior primarily in terms of the behaviors that are run, and less in terms of the data variables used. Moreover, callback functions are vital for supporting **interactivity**: many built-in JavaScript functions take in a callback function that specifies what should occur at some specific time (e.g., when the user clicks a button).

Often a callback function will be defined just to be passed into a single other function. This makes naming the callback somewhat redundant, and so it is more common to utilize **anonymous callback functions**:

```
//name anonymous function by assigning to variable
let sayHello = function(name){
    console.log("Hello, "+name);
}

function doWithWorld(funcToCall){
    funcToCall("world");
}
```

```
//pass the named function by name
doWithWorld(sayHello);

//pass in anonymous version of the function
doWithWorld(function(name){
    console.log("Hello, "+name);
});
```

- In a way, we've just "copy-and-pasted" the anonymous value (which happens to be a function) into the `doWithWorld()` call—just as you would do with any other anonymous variable type.
- Look carefully at the location of the closing brace `}` and parenthesis `)` on the last line. The brace ends the definition of the anonymous function value (the first and only parameter to `doWithWorld`), and the parenthesis ends *the parameter list* of the `doWithWorld` function. You need to include both for the syntax to be valid!
  - And since anonymous functions can be defined within other anonymous functions, it is not unusual to have lots of `})` lines in your code.

Closures

Functions are values, so not only can then be passed as parameters to other functions, they can also be *returned as results* of other functions!

```
//This function produces ANOTHER FUNCTION
//which greets a person with a given greeting
function makeGreeterFunc(greeting){
    //explicitly store the param as a local variable (for clarity)
    let localGreeting = greeting;

    //A new function that uses the `greeting` param
    //this is just a value!
    let aGreeterFunc = function(name){
        console.log(localGreeting+" "+name);
    }

    return aGreeterFunc; //return the value (which happens to be a function)
}

//Use the "maker" to create two new functions
let sayHello = makeGreeterFunc('Hello'); //says 'Hello' to a name
let sayHowdy = makeGreeterFunc('Howdy'); //says 'Howdy' to a name
```

```
//call the functions that were made
sayHello('world'); //"Hello world"
sayHello('Dave'); //"Hello Dave"
sayHowdy('world'); //"Howdy wold"
sayHowdy('partner'); //"Howdy partner"
```

- In this example, we've defined a function `makeGreeterFunc` that takes in some information (a greeting) as a parameter. It uses that information to create a new function `aGreeterFunc`—this function will have different behavior depending on the parameter (e.g., it can say "Hello" or "Howdy" or any other greeting given). We then return this new `aGreeterFunc` so that it can be used later (outside of the "maker" function).

  When we then call the `makeGreeterFunc()`, the result (a function) is assigned to a variable (e.g., `sayHello`). And because that result is a function, we can call it with a parameter! Thus `makeGreeterFunc` acts a bit like a "factory" for making other functions, which can then be used where needed.

The most significant part of this example is the *scoping* of the `greeting` variable (and its `localGreeting` alias). Normally, you would think about `localGreeting` as being scoped to `makeGreeterFunc`—once the maker function is finished, then the `localGreeting` variable should be lost. However, the `greeting` variable was in scope when the `aGreeterFunc` was created, and thus *remains in scope* (available) for that `aGreeteFunc` even after the maker function has returned!

This structure in which a function "remembers" its context (the in-scope variables around it) is called a **closure**. Even though the `localGreeting` variable was scoped outside of the `aGreeterFunc`, it has been *enclosed* by that function so it continues to be available later. Closures are one of the most powerful yet confusing techniques in JavaScript, and are a highly effective way of saving data in variables (instead of relying on global variables or other poor programming styles). They will also be useful when dealing with some problems introduced by Asynchronous Programming

11.4 Functional Looping

Another way that functional programming and callback functions specifically are utilized is to *replace loops with function calls.* For a number of common looping patterns, this can make the code more *expressive*—more clearly indicative of what it is doing and thus easier to understand. Functional looping was introduced in `ES5`.

To understand functional looping, first consider the common for loop used to iterate through an array of objects:

```
let array = [{...}, {...}, {...}];

for(let i=0; i<array.length; i++){
    let currentItem = array[i]; //convenience variable for current item

    //do something with current item
    console.log(currentItem);
}
```

While this loop may be familiar and fast, it does require extra work to manage the loop control variable (the `i`), which can get especially confusing when dealing with nested loops (and nested data structures are very common in JavaScript!)

As an alternative, you can consider using the Array type's `forEach()` method:

```
let array = [{...}, {...}, {...}];

//function for what to do with each item
function printItem(currentItem){
    console.log(currentItem;)
}

//print out each item
array.forEach(printItem);
```

The `forEach()` method goes through each item in the array and executes the given *callback function*, passing that item as a parameter to the callback. In effect, it lets you specify "what to do with each element" in the array as a separate function, and then "apply" that function to each elements.

- `forEach()` is a *built-in* method for Arrays—similar to `push()` or `indexOf()`. For reference, the "implementation" of the `forEach()` function looks something like:

  ```
  Array.forEach = function(callback) { //define the Array's forEach method
      for(let i=0; i<this.length; i++) {
          callback(this[i], i, this);
      }
  }
  ```

  In effect, the method does the job of managing the loop and the loop control variable for you, allowing you to just focus on what you want to do for each item.

- The *callback function* give to the `forEach()` will be executed with up to three argument (in order): (1) the *current item* in the array, (2) the *index* of the item in the array, and (3) the *array itself*. This means that your callback can contain up to three arguments, but since all arguments are optional in JavaScript, it can also be used with fewer—you don't need to include an argument for the index or array if you aren't utilizing them!

While it is possible to make a named callback function for `forEach()`, it is *much* more common to use an **anonymous callback function**:

```
//print each item in the array
array.forEach(function(item){
    console.log(item);
})
```

- This code can almost be read as: "take the `array` and `forEach` thing execute the `function` on that `item`".
- This is similar in usage to the enhanced for loop in Java.

Map

JavaScript provides a number of other functional loop methods. For example, consider the following "regular" loop:

```
function square(n) { //a function that squares a number
    return n*n;
}

let numbers = [1,2,3,4,5];  //an initial array

let squares = []; //the transformed array
for(let i=0; i<numbers.length; i++){
    let transformed = square(numbers[i]); //call our square() function
    squares.push(transformed); //add transformed to the list
}
console.log(squares); // [1, 4, 9, 16, 25]
```

This loop represents a **mapping** operation: it takes an original array (e.g., of numbers 1 to 5) and produces a *new* array with each of the original elements transformed in a certain way (e.g., squared). This is a common operation to apply: maybe you want to "transform" an array so that all the values are rounded or lowercase, or you want to *map* an array of words to an array of their lengths, or you want to *map* an array of values to an array of `<li>` HTML strings. It is possible to make all these changes using the above code pattern: create a new empty array, then loop through the original array and `push` the transformed values onto that new array.

However, JavaScript also provides a *built-in array method* called **`map()`** that directly performs this kind of mapping operation on an array without needing to use a loop:

```
function square(n) { //a function that squares a number
    return n*n;
}

let numbers = [1,2,3,4,5];  //an initial array

//map the numbers using the `square` transforming function
let squares = numbers.map(square);

console.log(squares); // [1, 4, 9, 16, 25]
```

The array's `map()` function produces a **new** array with each of the elements transformed. The `map()` function takes as an argument *a callback function* that will do the transformation. The callback function should take as an argument the element to transform, and **return** a value (the transformed element).

- Callback functions for `map()` will be passed the same three arguments as the callback functions for `forEach()`: the element, the index, and the array.

And again, the `map()` callback function (e.g., `square()` in the above example) is more commonly written as an anonymous callback function:

```
let numbers = [1,2,3,4,5];  //an initial array
let squares = numbers.map(function(item){
    return n*n;
});
```

Note: the major difference between the `.forEach` method and the `.map` method is that the `.map` method will **return** each element. If you need to create a new array, you should use `.map`. If you simply need to *do something* for each element in an array, use `.forEach`.

Filter

A second common operation is to **filter** a list of elements, removing elements that we don't want (or more accurately: only keeping elements that we *DO* want). For example, consider the following loop:

```
function isEven(n) { //a function that determines if a number is even
    let remainder = n % 2; //get remainder when dividing by 2 (modulo operator)
    return remainder == 0; //true if no remainder, false otherwise
}

let numbers = [2,7,1,8,3];  //an initial array

let evens = []; //the filtered array
```

```
for(let i=0; i<numbers.length; i++){
    if(isEven(numbers[i])){
        evens.push(numbers[i]);
    }
}
console.log(evens); //[2, 8]
```

With this **filtering** loop, we are *keeping* the values for which the `isEven()` function returns `true` (the function determines "what to let in" not "what to keep out"; a whitelist), which we do by appending the "good" values to a new array.

Similar to `map()`, JavaScript arrays include a *built-in method* called **`filter()`** that will directly perform this filtering:

```
function isEven(n) { //a function that determines if a number is even
    return (n % 2) == 0; //true if no remainder, false otherwise
}

let numbers = [2,7,1,8,3];  //an initial array

let evens = numbers.filter(isEven); //the filtered array

console.log(evens); //[2, 8]
```

The array's `filter()` function produces a **new** array that contains only the elements that *do match* a specific criteria. The `filter()` function takes as an argument *a callback function* that will make this decision. The callback function takes in the same arguments as `forEach()` and `map()`, and should return `true` if the given element should be included in the filtered array (or `false` if it should not).

- And again, we usually use *anonymous callback functions* for `filter()`:

  ```
  let numbers = [2,7,1,8,3];  //an initial array
  let evens = numbers.filter(function(n) { return (n%2)==0; }); //one-liner!
  ```

  (Since JavaScript ignores whitespace, we can compact simple callbacks onto a single line. ES6 and Beyond describes an even more compact syntax for such functions).

Because `map()` and `filter()` are both called on and produce arrays, it is possible **chain** them together, calling subsequent methods on each returned value:

```
let numbers = [1,2,3,4,5];  //an initial array

//get the squares of EVEN numbers only
let filtered = numbers.filter(isEven);
let squares = filtered.map(square);
```

```
console.log(squares); //[4, 16, 36]

//or in one statement, using results anonymously
let squares = numbers.filter(isEven)
                     .map(square);
console.log(squares); //[4, 16, 36]
```

This structure can potentially make it easier to understand the code's intent than using a set of nested loops or conditionals: we are taking `numbers` and then *filtering* for the evens and *mapping* to squares!

Reduce

The third important operation in functional programming (besides *mapping* and *filtering*) is **reducing** an array. Reducing an array means to *aggregate* that array's values together, transforming the array elements into a single value. For example, summing an array is a *reducing* operation (and in fact, the most common one!): it reduces an array of numbers to a single summed value.

- You can think of `reduce()` as a *generalization* of the `sum()` function found in many other languages—but rather than just adding (`+`) the values together, `reduce()` allows you to specify what operation to perform when aggregating (e.g., multiplication).

To understand how a *reduce* operation works, consider the following basic loop:

```
function add(x, y) { //a function that adds two numbers
    return x+y;
}

let numbers = [1,2,3,4,5];  //an initial array

let runningTotal = 0; //an accumulated aggregate
for(let i=0; i<numbers.length; i++){
    runningTotal = add(runningTotal, numbers[i]);
}
console.log(runningTotal); //15
```

This loop **reduces** the array into an "accumulated" sum of all the numbers in the list. Inside the loop, the `add()` function is called and passed the "current total" and the "new value" to be combined into the aggregate (*in that order*). The resulting total is then reassigned as the "current total" for the next iteration.

The *built-in array method* **`reduce()`** does exactly this work: it takes as an argument *a callback function* used to combine the current running total with the new value, and returns the aggregated total. Whereas the `map()` and `filter()` callback functions each usually took 1 argument (with 2 others optional), the `reduce()` callback function requires **2** arguments (with 2 others optional): the first will be the "running total" (called the **accumulator**), and the second will be the "new value" to mix into the aggregate. (While this ordering doesn't influence the summation example, it is relevant for other operations):

```
function add(x, y) { //a function that adds two numbers
    return x+y;
}

let numbers = [1,2,3,4,5];  //an initial array

let sum = numbers.reduce(add);
console.log(sum); //15
```

The `reduce()` function (not the callback, but `reduce()` itself) has a second optional argument *after* the callback function representing the initial starting value of the reduction. For example, if we wanted our summation function to start with `10` instead of `0`, we'd use:

```
//sum starting from 10
let sum = numbers.reduce(add, 10);
```

Note that the syntax can be a little hard to parse if you use an anonymous callback function:

```
//sum starting from 10
numbers.reduce(function(x, y){
    return x+y;
}, 10); //the starting value comes AFTER the callback!
```

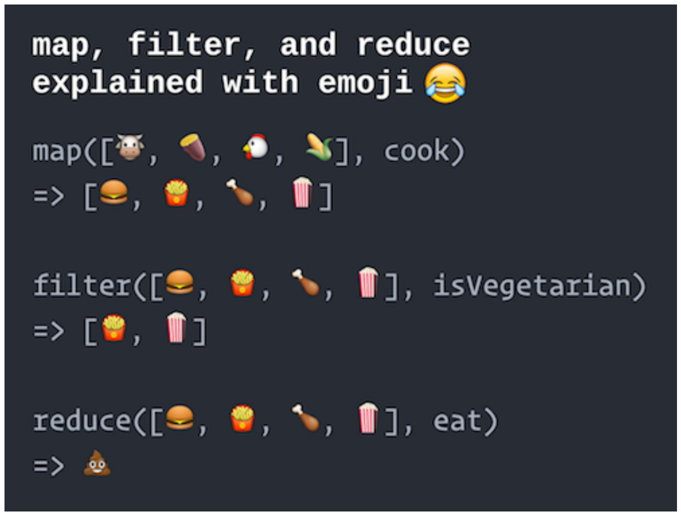
The *accumulator* value can be any type you want! For example, you can have the starting value be an empty object `{}` instead of a number, and have the accumulator use the current value to "update" that object (which is "accumulating" information).

To summarize, the `map()`, `filter()`, and `reduce()` operations work as follows:

All together, the **map**, **filter**, and **reduce** operations form the basic platform for a functional consideration of a program. Indeed, these kinds of operations are very common when discussing data manipulations: for example, the famous MapReduce model involves "mapping" each element through a complex function (on a different computer no less!), and then "reducing" the results into a single answer.

11.5 Pure Functions

This section was adapted from a tutorial by Dave Stearns.



map, filter, and reduce explained with emoji 😂

```
map([🐮, 🍠, 🐔, 🌽], cook)
=> [🍔, 🍟, 🍗, 🍿]

filter([🍔, 🍟, 🍗, 🍿], isVegetarian)
=> [🍟, 🍿]

reduce([🍔, 🍟, 🍗, 🍿], eat)
=> 💩
```

Figure 11.1: Map, filter, reduce explained with emoji. Not valid JavaScript syntax.

The concept of *first-class functions* (functions are values) is central to any functional programming language. However, there is more to the functional programming paradigm than just callback functions. In a fully functional programming language, you construct programs by combining small, reusable, **pure** functions that take in some inputted data, transform it, and then return that data for future use. *Pure* functions have the following qualities:

- They operate only on their inputs, and make no reference to other data (e.g., variables at a higher scope such as *globals*)
- They never modify their inputs—instead, they always return new data or a reference to an unmodified input
- They have no *side effects* outside of their outputs (e.g., they never modify variables at a higher scope)
- Because of these previous rules, they always return the same outputs for the same inputs

A functional program sends its initial input state through a series of these pure functions, much like a plumbing system sends water through a series of pipes, filters, valves, splitters, heaters, coolers, and pumps. The output of the "final" function in this chain becomes the program's output.

Pure functions are almost always easier to test and reason about. Since they have no side-effects, you can simply test all possible classes of inputs and verify that you get the correct outputs. If all of your pure functions are well-tested, you can then combine them together to create highly-predictable and reliable programs. Functional programs can also be easier to read and reason about because they end up looking highly *declarative*: it reads as a series of data transformations (e.g., take the data, then filter it, then transform it, then sort it, then print it), with the output of each function becoming the input to the next.

Although some functional programming zealots would argue that all programs should be written in a functional style, it's better to think of functional programming as another tool in your toolbox that is appropriate for some jobs, and not so much for others. Object-oriented programming is often the better choice for long-running, highly-interactive client programs, while functional is a better choice for short-lived programs or systems that handle discrete transactions (like many web applications). It's also possible to combine the two styles: for example, React components can be either object-oriented or functional, and you often use some functional techniques within object-oriented components.

- Indeed, the latest versions of Java—a highly object-oriented language—add support for functional programming!

If you are interested in doing more serious functional programming in JavaScript, there are a large number of additional libraries that can help support that:

- Lodash (and it's more pure variant lodash/fp)
- Ramda
- Lazy.js

There are also many languages that were designed to be functional from the get-go, but can be compiled down into JavaScript to run on web browsers. These include Clojure (via ClojureScript) and Elm.

Resources

- Functional Programming in JavaScript a *fantastic* interactive tutorial for learning functional programming in JS
- Higher Order Functions a chapter from the online textbook *Eloquent JavaScript*
- Scope & Closures an online textbook with an extremely detailed explanation of scoping in JavaScript

Chapter 12

Document Object Model (DOM)

The primary purpose of using JavaScript in a web page is to make that page **interactive**: the JavaScript language is used to program logical decisions that will effect what is shown on the page. It does this primarily by *changing the HTML rendered by the browser*. For example, JavaScript can be used to change the text inside a `<p>`, add addition `<li>` elements to a list, or to give a `<div>` a new CSS `class` attribute. The programmatic representation of the HTML elements currently being shown by the browser is known as the **Document Object Model (DOM)**. In web programming JavaScript code is used to modify the DOM (HTML elements) currently being shown by the browser in response to user input, thereby making the page interaction. This chapter introduces the Document Object Model and how to use JavaScript to manipulate it through user-driven interaction.

12.1 The DOM API

As you should recall from Chapter 3, HTML elements can be nested, allowing us to consider a webpage as a **"tree"** of elements:

- A **tree** is a hierarchical data structure, where each element (called a *node*) contains references to *child* elements. Following the arboreal metaphor, the "start" of the tree is called the *root note*, hierarchical sequences of nodes are called *branches*, and a *node* that does not have any children is called a *leaf*.

Considering a web page's content to be a tree of HTML elements is one way to *model* (represent) the structure of that information. This particular model

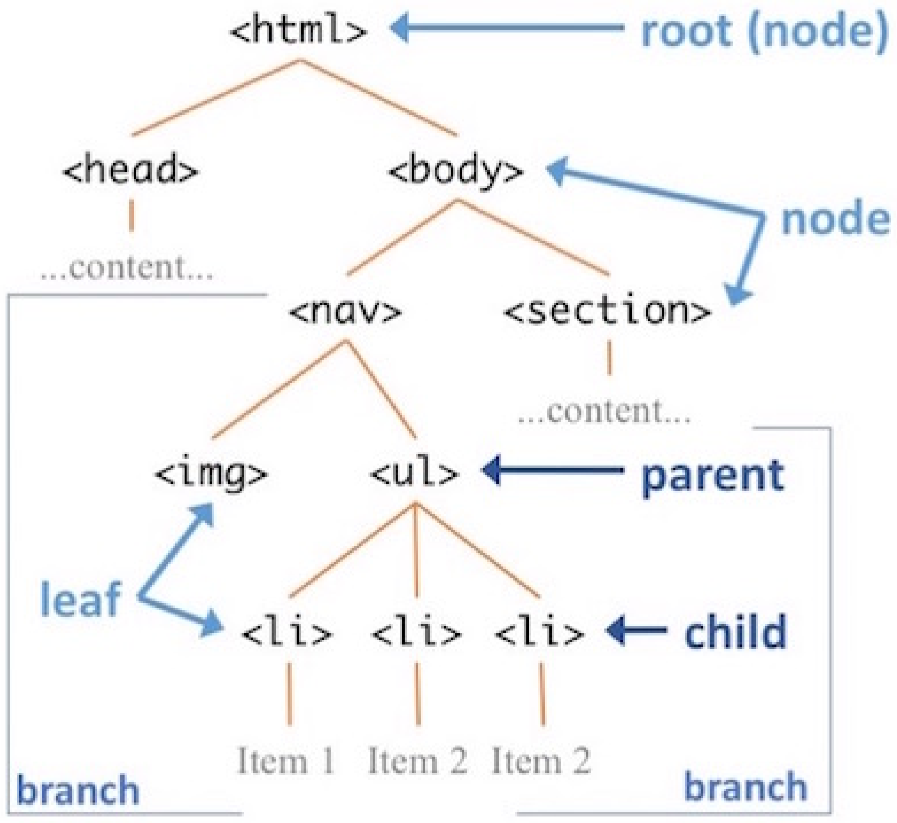


Figure 12.1: An example DOM tree (a tree of HTML elements).

of a web *document* (as a tree of *object* nodes) is called the **Document Object Model**, or **DOM** for short. In many ways the DOM *is* the HTML (though the HTML rendered in the browser, not the `.html` source code you've written)! Thus we can refer to the web page's content as "the DOM", and an HTML element as a "DOM element".

- Note that even "plain text" content (e.g., what is inside a `<p>` tag) are considered nodes in the DOM tree—they are "text content" nodes (instead of "element nodes").

Moreover, the DOM also provides an **Application Programming Interface (API)** which allows computer *applications* to *programmatically* (e.g., through JavaScript code) *interact* with it: accessing and manipulating the tree of elements. As you may recall from previous courses, an API is often a set of *functions* and *variables* that can be used give instructions to a program. The DOM API is no different: it is a group of functions you can call and variables (usually Object properties) you can adjust to change the rendered web content. You write code to call these functions in order to make a page interactive.

Global Variables

You can programmatically access the API in JavaScript by utilizing a set of global variables. **Global variables** are variables that are "globally" scoped: they are available anywhere in the program (not just within a particular function).

An important programming style rule is to **minimize** the use of global variables. Try to avoid creating too many new globals yourself!

Global variables in JavaScript are almost always *Objects* that have methods as their values. For example, the JavaScript language itself provides a global **`Math`** object that has includes a number of function properties (e.g., `sqrt()`, `floor()`, etc.).

```
console.log( typeof Math );  //=> 'object'
console.log( typeof Math.sqrt );  //=> 'function'
console.log( Math.sqrt(25) ); //=> 5
```

- In fact, the `console` object is *another* global variable provided by the JavaScript runtime (whether inside the browser or inside Node.js)!

The *web browser* also provides a number of global variables that you can use. For example **`window`** is a global object that represents the browser itself. You can use this object to get information about the browser:

```
/* example properties */
let width = window.innerWidth;   //viewport width
let height = window.innerHeight; //viewport height
```

```
var url = window.location.href; //url for this page

/* example functions */
window.alert("Boo!"); //show a popup alert. Do not use this.
window.scrollTo(0, 1000); //scroll to a position
window.setTimeout(callback, 1000); //execute callback after an delay
window.setInterval(callback, 1000); //execute callback repeatedly after interval
```

While these examples are included for completeness, most `window` functions are rarely used and should be avoided. Popups with the `window.alert()` function are inelegant, interrupt the user's actions, and produce a bad user experience—you should instead use in-window alerting options instead (such as showing a `<p class="alert">`). Browser control functions such as `scrollTo()` are non-standard and can vary drastically across systems and platforms. Proceed with caution when using `window` functions!

12.2 DOM Manipulation

While `window` represents the Browser, the **DOM** itself is represented by the **`document`** global object—`document` *is* the DOM (the *current* HTML rendered in the browser). You access properties and call methods of this object in order to manipulate the content displayed in the browser!

Referencing HTML Elements

In order to manipulate the DOM elements in a page, you first need to get a *reference* to the element you want to change—that is, you need a variable that refers to that element. You can get these variable references by using one of the `document` "selector" functions:

```
//element with id="foo"
let fooElem = document.getElementById('foo');

//elements with class="row"
let rowElems = document.getElementsByClassName('row'); //note the plural!

//<li> elements
let liElems = document.getElementsByTagName('li'); //note the plural!

/*easiest to select by reusing CSS selectors! */
let cssSelector = 'header p, .title > p';  //a string of a CSS selector

//selects FIRST element that matches css selector
```

```
let elem = document.querySelector(cssSelector);

//matches ALL elements that match css selector
let elems = document.querySelectorAll(cssSelector);
```

- The `document.querySelector()` is *by far* the most flexible and easy to use of these methods: it can easily do the same as all the other methods (just put in an id, class, or element selector). **You should always use `querySelector()`**.
- Note that the methods that return multiple nodes (e.g., `querySelectorAll`) return a `NodeList` object. While this is like an array (you can access elements via index through bracket notation and it has a `.length` property), it is **not** an array: meaning it doesn't support methods like `forEach()` and `map()` across all browsers. If you need to iterate through a `NodeList`, you should use a regular `for` loop. But in practice, you're much more likely to only work with single elements at a time.

Modifying HTML Elements

Once you have a reference to an element, you access properties and call methods on that object in order to modify its state in the DOM—which will in turn modify how it *currently* is displayed on the page. Thus by modifying these objects, you are dynamically changing the web page's content!

Important: setting these properties do not change the `.html` source code file! Instead, they just change the *rendered DOM elements* (think: the content stored in the computer's memory rather than in a file). If you refresh the page, the content will go back to how the `.html` source code file specifies it should appear—unless that also loads the script that modifies the DOM. What is shown on the page is the HTML with the JavaScript modifications added in.

Changing Content

You can use JavaScript to access and modify the **content** of a DOM element (e.g., the stuff between the start and close tags):

```
//get a reference to the FIRST <p> element
let elem = document.querySelector('p');

console.log(elem); //to demonstrate

let text = elem.textContent; //the text content of the elem
elem.textContent = "This is different content!"; //change the content
```

```
let html = elem.innerHTML; //content including HTML
elem.innerHTML = "This is <em>different</em> content!"; //interpreted as HTML
```

The `textContent` property of the element refers to *all* of the content, but considered as "plain text" this means that it is considered a "safe" property: you can assign strings that contain's HTML code (e.g., `<em>Hello</em>`), but that code will be escaped and not interpreted as HTML (instead the `<` and `>` will be written out as if you had used HTML entities). The `.innerHTML` property, on the other hand, is "not safe": any HTML included in the String you assign to it will be converted into DOM elements. This makes it not a great property to use unless unless you are absolutely certain the content came from a trusted source.

- The `innerHTML` property should be used primarily for including *inline* elements such as `<em>` or `<strong>`. For more complex HTML content, it is cleaner code (separation of concerns!) to explicitly create new elements—see below for details.

- You can "clear" the content of an element by setting it's content to be an empty string (`''`):

  ```
  let alertElem = document.querySelector('.alert');
  alertElem.textContent = ''; //no more alert!
  ```

Changing Attributes

You can also change the **attributes** of individual elements. Each attribute defined in the HTML specification is typically exposed as a *property* of the element object:

```
//get a reference to the `#picture` element
let imgElem = document.querySelector('#picture');

//access the attribute
console.log( imgElem.src ); //logs the source of the image

//modify the attribute
imgElem.src = 'my-picture.png';
```

You **cannot** access `element.class` or `element.style` attributes directly in this way; see below for specifics on changing the CSS of an element.

You can alternatively modify element attributes by using the methods `getAttribute()` (passing it which attribute to access) and `setAttribute()` (passing it which attribute to modify and what value to assign to that attribute):

```
let imgElem = document.querySelector('#picture');
imgElement.setAttribute('src', 'my-other-picture.png'); //set the src

console.log( imgElem.getAttribute('src') ); //=> 'my-other-picture.png'

//the `hasAttribute()` method returns a boolean.
let isThick = document.querySelector('svg rect')
                      .hasAttribute('stroke-width'); //chained anonymous variables
```

These methods will let you interact with attributes that are *not* defined by the HTML spec, such as `data-` attribute. However, they *don't* work with certain element attributes (such as the `value` attribute of an `<input>` element). Other elements may have their own special DOM properties: see the DOM Documentation for a list of HTML interfaces.

12.2.0.1 Changing Element CSS

It is possible to modify the **CSS classes** (and even inline styling) of an element. But rather than using the `class` property like with othe attributes, you instead access the **`className`** property. On modern browsers (IE 10 or later), this property supports methods `.add()` and `.remove()` for adding and removing classes from the list:

```
//access list of classes
let classList = elem.classList;

//add a class
elem.classList.add('small'); //add a single class
elem.classList.add('alert','alert-warning'); //add multiples classes (not on IE)

//remove a class
elem.classList.remove('small');

//"toggle" (add if missing, remove if present)
elem.classList.toggle('small');
```

- While IE 10+ does support these methods, it doesn't support *multiple arguments* for them (so you can't add multiple classes in a single method call). If you need to support older browsers (including any version of IE), you can instead modify the `.className` property as if it were a String:

  ```
  //fallback for IE (all)
  var classes = elem.className;
  classes += ' '+ 'sweet sour'; //modify the string (append!)
  elem.className = classes;     //reassign
  ```

The `classList` methods work perfectly on Microsoft Edge.

It is also possible to access and modify individual CSS properties of elements through the DOM element's `style` property. `.style` references an Object whose keys are the CSS property names (but written in *camelCase* instead of *kabob-case*)

```
let h1 = document.querySelector('h1');
h1.style.color = 'green';
h1.style.backgroundColor = 'black'; //not `.background-color`
```

In general, you should modify element CSS by changing the class of the element, rather than specific style properties.

Modifying the DOM Tree

In addition to modifying the individual DOM elements, it is also possible to access and modify the *DOM tree itself!* That is, you can create new elements and add them to the tree (read: webpage), remove elements from the tree, or pluck them out of the tree and insert them somewhere else!

First, note that each JavaScript DOM element has *read-only* properties referring to its parent, children, and sibling elements:

```
<main>
    <section id="first-section">
        <p>First paragraph</p>
        <p>Second paragraph</p>
    </section>
    <section id="second-section"></section>
<main>
```

```
//get reference to the first section
let firstSection = document.querySelector('#first-section');

//get reference to the "parent" node
let main = firstSection.parentElement;
console.log(main); //<main>...</main>

//get reference to the child elements (2 paragraphs)
let paragraphs = firstSection.children;
console.log(paragraphs.length); //2
console.log(paragraphs[0]); //<p>First paragraph</p>

//get reference to the the next sibling
let sectionSection = firstSection.nextElementSibling;
console.log(secondSection); //<section id="second-section"></section>
```

- Note that these properties only deal with *HTML elements*—text content nodes are ignored. You can instead use equivalent properties `parentNode` and `childNodes` to also consider text content nodes.

  SVG content doesn't support `parentElement`, but does support `parentNode`.

You can also call methods to create and add new HTML DOM elements to the tree. The `document.createElement()` function is used to create a new HTML element. However this element is *not* created a part of the tree (after all, you haven't specified where it would put into the page)! Thus you need to also use a method such as `appendChild` to add that new element as a child of another element:

```
//create a new <p> (not yet in the tree)
let newP = document.createElement('p');
newP.textContent = "I'm new!";

//create Node of textContent only (not an HTML element, just text)
let newText = document.createTextNode("I'm blank");

let main = document.querySelector('main');
main.appendChild(newP); //add element INSIDE (at end)
main.appendChild(newText); //add the text inside, AFTER the <p>

//add anonymous new node BEFORE element. Parameters are: (new, old)
main.insertBefore(document.createTextNode("First!"), newP);

//replace node. Parameters are: (new, old)
main.replaceChild(document.createTextNode('boo'), newText);

//remove node
main.removeChild(main.querySelector('p'));
```

The `appendChild()` method is considered a cleaner approach than just modifying the `innerHTML` property, as it allows you to adjust the DOM tree without erasing what was previously there. A common practice is to use `document.createElement()` to create a *block* element, then set the `innerHTML` of that element to its content (which can include *inline* elements), and then use `appendChild` to add the new block element to the tree at the desired location.

Accessibility

Whenever you learn a new technology, you should ask: **how does this affect accessibility?** With the JavaScript code modifying the rendered DOM, it is possible that the content of a page will change *after* it has been read by a screen reader. And while a sighted user will likely be able to see the change visually, a screen reader has no way of knowing that something on the page is different unless you tell it.

You can let screen readers know that an element in a page may have its content change *in the future* by making that element into an ARIA Live Region. Live regions are "watched" by assistive technologies, and whenever the content changes they will speak the new content to the reader as if it were being read for the first time.

You make an element into a live region by giving it the `aria-live` attribute:

```html
<div aria-live="polite">
  This content can change!
</div>
```

The value assigned to the `aria-live` attribute is the "politeness level", which specifies the priority by which the screen reader should read the change. The most common option (that you should almost always use) is `"polite"`, which indicates that the changed text will be read only once the user has *paused* whatever is currently being read. A `"polite"` alert doesn't interrupt the currently being read text or description, but instead will be injected when there is a break (if the current reading goes on for too long, then the new content will not be spoken).

- The other option is `"assertive"`, which indicates that the new content should be spoken as soon as it changes, possibly interrupting other content. This should only be used for important information (like alerts, warnings, or errors), as it can interrupt the user's flow in ways that are very disorienting. In short: *always be polite!*

12.3 Listening for Events

In order to make a page **interactive** (that is, able to change in response to user actions), you need to be able to respond to *user events*. Whenever a user interacts with a computer, the operating system announces that interaction as an **event**—the *event* of a button being clicked, the *event* of the mouse being moved, the *event* of a keyboard key being pressed, etc. These events are **broadcast** to the entire system, allowing any application (including the browser) to "respond" the occurrence of the event, such as by executing a particular JavaScript function.

Thus in order to respond to user actions (and the *events* those actions generate), we need to define a function that will be executed **when the event occurs**. You will define a function as normal, but the function will not get called by you as a particular step in your script. Instead, the function you specify will be executed *by the system* when an event occurs, which will be at some indeterminate time in the future. This process is known as event-driven programming. It is also an example of **asynchronous programming**: in which statements are not executed in a single order one after another ("synchronously"), but may occur "out of order" or even at the same time! (For more about working with asynchronous programming, see Chapter 14).

In order for your script to respond to user events, you need to *register an event listener*. This is a bit like following someone on social media: you specify that you want to "listen" for updates from that person, as well as what you want to do when you "hear" some news from that person.

- Specifying that you want Slack to notify you when your name is mentioned is another good analogy!

The DOM API allows you to register an event listener by call the **`.addEventListener()`** on a selected element (e.g., on the element that you want to listen to). This method takes two arguments: a string representing what kind of event you want to listen for, and a *callback function* to execute when you hear that event:

```
//a (named) callback function
function onClickCallback() {
    console.log("You clicked me!");
}

//get a reference to the element we want to "listen" to
let button = document.querySelector('button');

//register a listener for 'click' events
button.addEventListener('click', onClickCallback);
```

- When the button is clicked by the user, it will "shout" out a `'click'` event ("I was clicked! I was clicked!"). Because you have set up a listener (an alert/notification) for such an occurrence, your script will be able to do something—and that something that it will do is run the specified callback function.

  It's like you handed someone a recipe and told them "when I call you, bake this cake!"

- It is **much** more common to use an *anonymous function* as the callback:

  ```
  let button = document.querySelect.select('button');
  button.addEventListener('click', function() {
      console.log("You clicked me!");
  });
  ```

- Note that this listener *only* applies to that particular button—if you wanted to respond to a different button, you'd need to register a separate listener! Also, as the method name implies, it is possible to add multiple listeners (callbacks) to the same element for the same event: all of them will be executed "at once".

The event callback will be passed in a single argument: an object representing the *"event"* that occurred. (Since all parameters are optional in JavaScript, and it wasn't used in the above example, it wasn't included in the callback definition). This event includes information such as where the event occurred (in x,y coordinates), what DOM element caused the event, and more:

```
elem.addEventListener('click', function(event) {
   //get who was clicked;
   let clickedElem = event.target; //target property of the event
   console.log(clickedElem);
});
```

Also note that sometimes you want to stop the "normal" results of an event from occurring. For example, perhaps you don't want a button to do it's normal button thing (such as submitting a form), and instead want to provide your own custom behavior. To support this, you can "interrupt the event" by calling the following methods on the event:

```
submitBtn.addEventListener('click', function(event) {

  event.preventDefault(); //don't do normal behavior
  event.stopPropagation(); //don't pass the event to parents

  //..do custom behavior here

  return false; //don't do normal behavior OR propagate! (for IE)
})
```

Types of Events

There are numerous different events that you can listen for, including:

- Mouse Events such as **`'click'`**. The `event.offsetX` and `event.offsetY` will provide (x,y) coordinates for the clicks location *relative to the target element*; you can use `clientX/Y` for coordinates relative to the browser window, or `pageX/Y` for coordinates relative to the document (regardless of scrolling). See this post for details, and this page for an example.

  Other mouse events include `'dblclick'` (double-click), `'mousedown'` (mouse button is pressed down, may be held), `'hover'` (mouse hover), `'mouseenter'` (mouse moves over element), `'mousemove'` (mouse moves over element), and `'mouseleave'` (mouse moves of of element).

- Keyboard Events such as **'keydown'**. The `event.key` property is used to determine *which key* was pressed, giving a predefined key value you can check:

  ```
  elem.addEventListener('keydown', function(event){
      if(event.key === 'ArrowUp'){
          console.log("Going up!")
      }
      //...
  });
  ```

  The `event` object also has properties to check if any "modifier keys" such as shift, control, or meta (Windows/command) are held when the event occurs.

  Note that you almost always want to respond to the `'keydown'` and `'keyup'` events; the `'keypressed'` event is sent later and only applies to non-modifier keys.

- Window Events are event created by the `window` global, which we are also able to register event listeners on! For example, the `'resize'` event can be used to identify when the window has changed size (e.g., if you want to make the content responsive as well as the CSS):

  ```
  window.addEventListener("resize", function() {
      //...
  });
  ```

  (See the documentation for advise on using this callback)

  Additionally, the `window` global defines a special event callback that occurs when the web page has finished loading. You can assign your own function to this callback to run code only *after* the webpage has loaded (e.g., for scripts specified in the `<head>`):

  ```
  window.onload = function() {
      //...do stuff once page is ready (e.g., run the rest of your code)
  }
  ```

Style guideline: always register event listeners in the JavaScript—do *not* utilize the HTML attributes such as `onclick`. This is to help keep concerns separated: the HTML should not need to know anything about the JavaScript that is utilized (since the browser may not even support JavaScript!), but it's okay for the JavaScript to rely on and modify the HTML.

Event-Driven Programming

In a typical web program event callback functions can occur repeatedly, over and over again (e.g., every time the user clicks a button). This makes them potentially act a bit like the body of a `while` loop. However, because these callbacks are *functions*, any variables defined within them are **scoped** to that function, and will not be available on subsequent executions. Thus if you want to keep track of some additional information (e.g., how many times the button was clicked), you will need to use a variable declared *outside* of the function (e.g., a **global**). Such variables can be used to represent the **state** (situation) of the program, which can then be used to determine what behavior to perform when an event occurs, following the below pattern:

```
//pseudocode
WHEN an event occurs {
   check the STATE of the program;
   DECIDE what to do based on that state;
   UPDATE the state as necessary for the next event;
}
```

For example:

```
let clickCount = 0;  //keep track of the "state" (global)
document.querySelector('button').addEventListener('click', function() {
    if(clickCount > 10) {  //decide what to do
        console.log("I think you've had enough");
    }
    else {
        clickCount++;  //change state (+1)
        console.log('You clicked me!');
    }
});
```

- These "state" variables can be global, or can simply be declared within a containing function as a closure. State variables are often objects, with individual values stored as the properties. This provides a name-spacing feature, and helps to keep the code from being cluttered with many variables.

Resources

- What is the DOM? (CSS-Tricks)
- Document Object Model reference (MDN) complete DOM reference
- JavaScript HTML DOM reference (w3c)
- Introduction to Events (MDN)

Chapter 13

JavaScript Libraries

Many web programmers encounter the same programming requirements as they develop interactive web sites. For example, many websites will need to wrangle and manipulate objects and arrays, show standard components (such as modal windows or collapsible content), or do similar forms of complex DOM manipulation. Since one of the main principles of software development is ***reuse***, developers will often make the code solving these problems available for others to use in the form of **libraries and frameworks**. These are publicly release script files that you can download and include in your project, allowing you to more quickly and easily develop complex applications. This is the amazing thing about the open-source community: people solve problems and then make those solutions available to others.

Modern web applications make *extensive* use of external libraries, whether by integrating many different libraries into a single app, or by relying on a particular framework (which may itself be comprised of many different libraries)! This chapter describes how to include and utilize external JavaScript libraries in your web page, and presents the popular jQuery library as an example.

- For a sense of scale, the **npm** package manager's directory lists almost *half a million* different JavaScript libraries!

Note that external scripts are generally referred to as either *libraries* or *frameworks*. However, these terms are not entirely interchangeable. A **library** is a set of behaviors (functions) that you are able to utilize and call within your code. For example, `Lodash` (described below) is a library that provides utility functions you can use. You call a library's code at your whim. A **framework**, on the other hand, provides a set of code into which you insert your *own* behaviors, either by subclassing provided components or by specifying your own callback functions. The framework calls *your* code at its whim. Martin Fowler refers to this as an *"Inversion of control"*. Frameworks often seem to be easier to use (they do more with less work on your part!), but can be hard to customize to achieve your exact goals. Libraries may be harder to use, but can likely be deployed exactly as needed. This chapter focuses mainly on *libraries*; React (detailed in later chapters) is more of a framework.

13.1 Including a Library

Just like you can include multiple CSS files in a page with multiple `<link>` elements, you can include multiple JavaScript scripts with multiple `<script>` elements.

```
<script src="alpha.js"></script> <!-- run this script first -->
<script src="beta.js"></script> <!-- run this script second -->
```

Importantly, script files are executed in the order in which they appear in the DOM. This matters because all scripts included via the `<script>` element are all run within the same namespace. This means variables and functions declared in one script file are also available in later scripts— it's almost as if all the script files have been combined into a single file.

```
/* alpha.js */
let message = "Hello World";
```

```
/* beta.js */
console.log(message); //=> "Hello World"
                      //variable was defined in previous script!
```

The order matters: if `beta.js` were included *before* `alpha.js`, then when it is run the `message` variable won't have been defined yet, causing an error.

Other techniques for managing and organizing large numbers of script files will be discussed in Chapter 15.

Like CSS frameworks, JavaScript libraries are thus included in a page by providing a `<script>` tag that references the JavaScript file that contains that library's code. For example, you can include the Lodash of helpful data processing functions:

```
<body>
    ...content

    <!-- include JavaScript files -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.4/lodash.min.
    <script src="js/my-script.js"></script>
</body>
```

- Like CSS frameworks, JavaScript libraries can be accessed by referencing the file on a **CDN**, by downloading the source code directly (check for the `.js` file; you may need to unpack a `.zip` file), or by installing the library via a package manager like `npm`. Refer to Chapter 3 for details—just change `.css` to `.js`!

- Like CSS, JavaScript files can be *minimized* (usually named as `.min.js` files). Minimizing JavaScript files not only removes whitespace, but also often replaces local variable names with shorter, meaningless ones such as `a` or `x`.

You **always** want to load external libraries before your script, so that any variables and functions they define will be available to your code! If you put your script first, then those functions won't be available yet!

Because all JavaScript files share the same namespace, most popular libraries make all of their functions available as properties of a single **global variable**. For example, Lodash provides a global variable called `_` (a single underscore—a "low dash"). You use this variable to call the provided functions:

```
let array = [3,4,3];

let unique = _.uniq(array); //call function of lodash object

console.log(unique); //[3,4], the unique elements!
```

- Once the library is loaded, the globals it provide can be accessed just like built-in globals `window`, `document`, `Math`, etc.
- Lodash uses an otherwise silly variable name so that it's quick and easy to type, and to avoid conflicts with other variables you might create.

Lodash provides lots of useful functions that can be called on the global _ object. For example:

- `_.random()` to generate a random number within a range
- `_.range()` to generate an array of numbers within a range
- `_.includes()` to determine if a value is in an array (cross-browser!)
- `_.pick()` to produce a new object that has only select properties of another
- `_.merge()` to combine the properties of two objects into a single new object

Note that exactly what global object is provided (and what functions it has) is different for every different library. You need to look through the documentation (often the `README.md` file on Github) for instructions and examples on how to call and utilize that library.

- "Knowing" a library or framework is just about being familiar with what functions you can call from it!

Pro-tip: Visual Studio Code will automatically download the type definitions for any library listed as a dependency in the project's `package.json`. For example, if you include Lodash as a dependency (by installing it with `npm` `install --save lodash`), then just typing the `_.` will cause VS Code to provide autocomplete options. This makes it a good idea to install JavaScript libraries via `npm`, even if you plan on loading the library from a CDN instead of from a local file.

13.2 Example: jQuery

As an extended example of a JavaScript library, the remainder of this chapter describes **jQuery**—a popular JavaScript library that helps with DOM manipulation. jQuery was designed to provide easier, faster, and more reliable methods for manipulating the DOM in the ways described in the previous chapter. jQuery is one of the most popular libraries used in web development, and is used in around 70% of the most popular websites.

jQuery was developed to help fill in gaps in the JavaScript language and to standardize functionality across browsers. It made it so that you could write manipulate the DOM while writing 10% of the code, and produce scripts that would run on both Firefox *and* Internet Explorer!

However, in the 10+ years since it's release, later versions of the JavaScript language and the DOM API have improved (e.g., by introducing a `querySelector()` function) so that much of what made jQuery special is now standard and widely supported. Indeed, jQuery' approach to interactive webpages is now common practice. Nevertheless, jQuery is still commonly used in web systems (Bootstrap's JavaScript components utilize jQuery), and is popular enough that it you should be familiar with it in the likely chance that you need to engage with a web system build more than 3 years ago.

As with other libraries, you include jQuery by referencing it from a `<script>` tag in your HTML (*before your own script!*):

```html
<script src="http://code.jquery.com/jquery-3.2.1.min.js"></script>
```

- Note that version jQuery `3.x` is for current browsers. If you need to support an older version (e.g., IE 6-8), you can use version `1.12`. Yes, jQuery supports IE 6!

Loading the jQuery library creates a **`jQuery`** variable in the global scope, meaning you can access this variable and utilize it in your script.

The `jQuery` variable is actually a *function* that you can call, called the **Selector function**. This function is used to select DOM elements by CSS selector, similar to how `document.querySelector()` works:

```js
//selects element with id="foo" (e.g., <p id="foo">)
let fooElem = jQuery('#foo');
```

```
//selects all <a> elements
let allLinksArray = jQuery('a');
```

However, it is much more common to use a provided *alias* for the `jQuery()` function called **`$()`**. This "shortcut" lets you select elements with a single character (instead of 6)!

```
//selects element with id="foo" (e.g., <p id="foo">)
let fooElem = $('#foo');

//selects all <a> elements
let allLinksArray = $('a');
```

- Similar to Lodash's `_`, jQuery uses `$` because no one in their right might would name a variable that, so the chance of having a namespace conflict is minimal.

Like `document.querySelector()`, the jQuery selector function handles most all selectors you are familiar with from CSS, as well as some additional useful pseudo-classes:

```
$('#my-div') // by id
$('div') // by type
$('.my-class') // by class
$('header, footer') // group selector
$('nav a') // descendant selector
$('p.red') // scoped selector

$('section:first') // first <section> element
                   // (not a css selector!)
```

Maniputing the DOM

Once you've selected some elements, jQuery provides *methods* that perform most of the manipulations you would do using DOM properties:

```
var txt = $('#my-div').text();      //get the textContent
$('#my-div').text('new info!');     //change the textContent

$('#my-div').html('<em>new html!</em>'); //change the HTML

$('svg rect').attr('height');       //get attribute (of all selected)
$('svg rect').attr('height',200);   //set attribute (of all selected)
$('svg rect').attr( {x:50, y:60} ); //set multiple attributes by passing in an object

$('section').addClass('container'); //add class (to all selected)
$('section').removeClass('old');    //remove class (from all selected)
```

```
$('body').css('font-size','24px');  //set css property (of all selected)
$('body').css( {'font-size':'24px',
                'font-family':'Helvetica'} ); //set multiple properties via an objec
```

If you compare this to the equivalent operations using the DOM, you'll notice that (a) jQuery is much more concise. jQuery is also more powerful: for example, you can set multiple attributes with a single call to `.attr()` by passing in an *object* containing the attributes you wish to set (the keys are the attribute names). This lets you easily change lots of attributes at once!

Importantly, note that all of these methods apply the change to ***all*** elements selected by the jQuery selector function. You do not need to use a loop to apply changes to multiple elements; you can just select the elements you wish to modify and change them all at once. This does mean that you may need to be careful about your selector—only select the elements you actually wish to work with!

jQuery also provides methods that allow you to manipulate the DOM tree (e.g., to add, move, or remove elements):

```
//create an element (not yet in the DOM)
let newP = $('<p class="new">This is a new element</p>'); //notice the tag!

//add content to DOM
$('main').append(newP); //add the element INSIDE <main>, at end
$('main').append('<em>new</em>'); //can a create element on the fly!

$('main').prepend('<em>new</em>'); //add new <em> element INSIDE <main>, at beginnin

$('main').before('<header>'); //insert new <header> BEFORE <main> (older sibling)
                              //notice you can omit the closing tag if no content

$('footer').insertAfter('main'); //insert selected (<footer>) AFTER parameter (<main
                                 //since the <footer> was selected, it will move!

$('main').wrap('<div class="container"></div>'); //surround the <main> with a .conta

$('footer').remove(); //remove selected <footer> element
$('main').empty(); //remove all child elements of <main>
```

The first important thing to note is that you create new elements by provided the HTML content (including the tag `<>`) to the `$()` function. This is distinct from `document.createElement()`, which explicitly does *not* include the `<>` angle brackets. If creating an element with no content, you can even just specify the start tag!

jQuery also provides much more powerful manipulation methods, allowing you to easily position elements inside, outside, around, and in place of other elements.

Although jQuery will allow you to create arbitrarily complex HTML elements to add to the DOM, you should **avoid** writing large chunks of HTML inside your `.js` file to append via jQuery (e.g., don't add an entire DOM tree at once!). This violates the separation of concerns, and makes your code difficult to read, interpret, and modify (because your HTML is now in two places!). If you do need to dynamically insert large amounts of "hard-coded" HTML, you should instead write that content inside the `.html` file, make it invisible (e.g., `display:none`), and then use jQuery to move and show the element when needed.

Handling Events

jQuery also provides *convenience methods* for registering event listeners:

```
$('#button').click(function(event) {
   console.log('clicky clicky!');

   //who was clicked
   let element = $(event.target);
});
```

- There are equivalent methods for other events: `.mousedown()` `.keypress()`, etc. If you want to listen for an event that jQuery doesn't provide a convenience method for, you can use the `.on()` function:

  ```
  $('#search-input').on('input', callback);
  ```

By using a method such as `.click(callback)` rather than `.addEventListener('click', callback)` or event `.on('click', callback)`, you avoid potential hard-to-catch bugs that may get introduced from misspelling `'click'`—rather than having the browser listen for `'clik'` events and then never seeming to respond to your actions, it will instead report an error (`.clik()` is not a known function!)

The `event` parameter passed into the `.click()` function's callback is exactly like the `event` passed to callbacks of `addEventListener()` (though jQuery standardizes cross-browser quirks)—thus you can access the source of the event with the `event.target` property. However, this property refers to a DOM element, *not* a jQuery selector. DOM elements don't support the jQuery methods described above—those are only available to "jQuery selection objects". Thus if you want to work with a DOM element, you need to "select" it again using the `$()` function, thus providing a jQuery selection that has all the useful jQuery methods.

- Additionally, jQuery assigns the `this` variable inside an event callback to the event's target; this you can equivalently use `$(this)` to select the element.

The previous chapter noted the `window.onload` even listener, which was used to determine when the DOM had finished loading and so was ready to be manipulated. jQuery provides a similar functionality via the `.ready()` listener (usually called on the whole `document`):

```
$(document).ready(function() { //this need not be an anonymous function
    //program goes here
    console.log('Hello world!');

    //...
});
```

This is a very common pattern: often with jQuery entire programs will be specified inside the `.ready()` callback, so that the `<script>` tags can be placed in the `<head>` and downloaded quickly but still run only when the DOM is available. This pattern is *so* common in fact, that the jQuery selector function can serve as a shortcut to it:

```
//equivalent to the above
$(function() {
    //program goes here
    console.log('Hello world!');

    //...
});
```

- If you pass a *function* rather than a Selector string to the `$()` jQuery function, it will be interpreted as specifying a callback function to run when the document is ready! This is something to be aware of if you look at someone else's code and see it just start with a random `$(function)`.

And more!

This is only the tip of the iceberg for what jQuery can do. For example, jQuery also provides a number of utility functions that can be called on the `jQuery` (or `$`) global:

```
//check if an item is in an array
$.inArray(4, [3,4,3] );
jQuery.inArray(4, [3,4,3] ); //equivalent, but maybe easier to read

//find an item in an array that matches the filter function
//this is like .filter, but works on old browsers (if right jQuery version)
```

```
$.grep( [3,4,3], function(item) {
   return item > 3;
});

//iterate over arrays or objects -- works for either!
$.each( [3,4,3], function(key, value) {
   console.log('Give me a '+value);
});

$.each( {first:'Joel',last:'Ross'}, function(key, value) {
   console.log(key+' name: '+value);
});
```

- This can be useful, though JavaScript native functions or Lodash methods will often be easier and (computationally) faster.

But as a final fun example: jQuery also provides functions that allow you to easily produce animated effects!

```
$('#id').fadeIn(1000);   //fade in over 1 second
$('#id').fadeOut(500);   //fade out over 1/2 sec
$('#id').slideDown(200); //slide down over 200ms
$('#id').slideUp(500);   //slide up over 500ms
$('#id').toggle();       //toggle whether displayed

//custom animation syntax:
//$(selector).animate({targetProperties}, speed [, doneCallback]);
$("#box").animate({
    left: '500px', //make the box fly around!
    opacity: '0.5',
    height: '200px',
    width: '200px'
}, 1500);
```

Resources

- Lodash Documentation
- jQuery Documentation
- jQuery Learning Center
- jQuery Tutorial (w3c)

Chapter 14

AJAX Requests

JavaScript allows you to dynamically define the content of a web page, generating the DOM at runtime rather than in `.html` source files. One of the primary reasons we would want to dynamically produce a DOM is if the web page's content is based on some **data** that may change over time: for example, the kind of data that is available through a **Web API**. By using JavaScript to render the DOM, you can quickly produce large amounts of HTML needed to display large data sets, sure that you have up-to-date data each time the page loads, and even *automatically refresh the page content* without requiring the user to reload!

This chapter describes how to use JavaScript to dynamically send HTTP Requests to download data (without reloading the page!), as well as how to perform the **asynchronous programming** needed when working with web requests and other time-consuming operations.

Note that this lecture assumes that you have a basic familiarity with RESTful Web APIS, including how to read and access their endpoints. For a review of some of the terminology used in APIS and RESTful requests, see the INFO 201 course reader.

14.1 AJAX

As discussed in Chapter 2, you download data from the Internet by sending an **HTTP Request** and then processing the **response**. In everyday usage, HTTP Requests are normally sent by the *browser* when the user enters a URL or clicks on a link. By default, if you wanted to download new data, you'd need to have the browser send a new request, loading a new page (or reloading the current page) in order to show that result.

To make modern dynamic web pages that display new data without needing to refresh the browser, we use a technique to send HTTP Requests *from JavaScript code* rather than from the browser. This allows us to "by-pass" the browser and get new data (and change the webpage) without reloading it! This technique is referred to as **AJAX** (**A**synchronous **J**avaScript **A**nd **X**ML)—we write code that sends an "request with AJAX" or an "AJAX request".

Fun fact: The technology used to send AJAX requests was originally developed by Microsoft in the late 90s to support their fledgling web version of the Outlook email/calendar app. The JavaScript functions used to send these requests were included in Internet Explorer as a *non-standard* feature—an example of a browser adding new functionality that it thinks will be useful but that doesn't work on other platforms. However, AJAX quickly gained popularity (particularly when Google showed off what you could do with it via Gmail and Google Maps), and has since become a standard that is now supported by all browsers. This is how standards come into existence!

XML and JSON

AJAX is called "AJA**X**" because it was originally designed to request data in XML format. **XML** (E**X**tensible **M**arkup **L**anguage) is a markup language (like HTML) that is used to encode meaning in content in a format that is both human *and* computer readable. The syntax for XML is *exactly* the same as HTML: in fact, HTML can be seen as a "subset" of the language. You can think of XML as "HTML, but you get to make up your own element names!"

```
<!-- Some XML encoding information about a person -->
<person>
   <firstName>Alice</firstName>
   <lastName>Smith</lastName>
   <favorites>
      <music>jazz</music>
      <food>pizza</food>
      <numbers>
          <item>12</item>
          <item>42</item>
      </numbers>
   </favorites>
</person>
```

- The XML language does not define any particular tags the way HTML does; instead it is up to individual applications to determine what tags it will recognize and interpret (and what tags it would see as gibberish)—what is referred to as a XML Schema.

At the time AJAX was first developed, XML was the most common way of encoding generic data for transmission. And because XML is a tree of elements just like the DOM, similar methods could be used to navigate and extract information from the tree. However, XML is a very *verbose* language: it requires a lot of characters to encode information (meaning that the amount of data being transferred is larger), and traversing an element tree requires a lot of code. As such, JavaScript developers (led by Douglas Crockford) developed an alternative language called **JSON** (**J**ava**S**cript **O**bject **N**otation) that is more compact than XML and can be *directly* parsed into JavaScript objects and arrays:

```
{
  "firstName": "Alice",
  "lastName": "Smith",
  "favorites": {
    "music": "jazz",
    "food": "pizza",
    "numbers": [12, 42]
  }
}
```

JSON format uses a syntax that is almost identical to that for defining Object literals in JavaScript, with a few key differences:

- JSON always defines an Object `{}` at the "top level".
- JSON object **keys** (which must be strings) *must* be written in double-quotes.
- JSON **values** can only be strings, numbers, booleans (`true` or `false`), arrays (`[]`), or other objects. You cannot include a function in JSON.
- JSON objects and arrays can't have trailing commas or other extraneous symbols—no comments!

The JavaScript language provides a global object `JSON` (like the global `Math` object) that can be used to convert from encoded *strings* of JSON content (e.g., the above code block as a single string variable `'{"firstName":"Alice"}'`) to JavaScript objects, and vice versa:

```
//convert from Object to encoded String
let personObj = {firstName:"Alice", lastName:"Smith", id:12} //JavaScript object
let personString = JSON.stringify(personObj); //turn object into JSON string
console.log(personString); //=> '{"firstName":"Alice","lastName":"Smith","id":12}'
console.log(typeof personString); //=> 'string'

//convert from encoded String to Object
let favoritesString = '{"music":"jazz", "numbers":[12,42]}'; //a string, not an object!
let favoritesObj = JSON.parse(favoritesString); //turn JSON string into object
console.log(favoritesObj); //=> { music: 'jazz', numbers: [ 12, 42 ] }
console.log(typeof favoritesObj); //=> 'object'
```

- Note that if your JSON string is not properly formatted (e.g., you're missing a quote), the `JSON.parse()` function will throw a `SyntaxError`.

> The exact error in the JSON string can be hard to find; online tools can help show the problem.

JSON has replaced XML as the encoding of choice for working with AJAX requests—however, the technique is still referred as "AJAX" ("AJA*J*" isn't as easy to say!)

14.2 Fetching Data

AJAX support is built into browsers through the included `XMLHttpRequest` global variable (the "xml http thing"). This object provides functions that allow you to send an HTTP request to the server, but the object's API is **really complex to use**:

An example `XMLHTTPRequest`

```
//create a new XMLHttpRequest object
let request = new XMLHttpRequest();

//configure it to do an HTTP GET request for some URL
request.open('GET', 'https://domain.com/data', true);

//add a listener for the "load" event (when the data has been downloaded)
request.addEventListener('load', function() {
    if (request.status >= 200 && request.status < 400) { //check response status
        let data = JSON.parse(request.responseText);
        console.log(data); //do something with the data
    }
});

//listen for "error" events if there was a network error
request.addEventListener('error', function() {
    //handle error...
})

//finally, send the request to the server!
request.send();
```

Instead of needing to understand all that code, developers tended to use functions from external libraries such as jQuery's `$.getJSON()` or `$.ajax()`:

```
$.getJSON('https://domain.com/data', function(data) {
    //`data` is the already-parsed JSON data
    console.log(data); //do something with the data
});
```

But this requires including the jQuery library in your page, and since the need for jQuery is rapidly going away, other options are now *built in* to modern browsers. In particular, we will utilize the **`fetch()`** API to easily send AJAX requests for data!

`fetch()` is an recent standard—so recent that it is not supported by IE (any) or Safari 10. However, we can still use `fetch()` with these browsers by including a **polyfill**—an external library that replicates an existing API in platforms that don't support it! The `fetch()` polyfill will provide a `fetch()` function to browsers that don't provide it (leaving other browsers unchanged) that uses the existing `XMLHttpRequest` without you needing to interact with that object.

It's easiest to just load the polyfill from a CDN:

```html
<!-- put this BEFORE your own script! -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/fetch/2.0.3/fetch.min.js"></script>
```

The `fetch()` function makes it easy to send a request: simply call it and pass in the URL of the data you wish to download:

```js
fetch('https://domain.com/data');
```

In some browsers, you will not be able to send an AJAX request when the web page is loading via the `file://` protocol (e.g., by double-clicking on the `.html` file). This is a security feature to keep you from accidentally running an HTML file that contains malicious JavaScript that will download a virus onto your computer. Instead, you should run a local web server such as `live-server` for testing AJAX requests.

14.3 Asynchronous Programming

However, the `fetch()` function does **NOT** directly return the data you want to download! Downloading data off the internet can take a long time: the network connection may be slow and the amount of data to download may be quite large (metadata for the latest 100 tweets from Twitter involves almost 500k of JSON content). Because fetching data may take time, AJAX requests are made **asynchronously** (that's the "A" in AJAX)—the download will occur *at the same time* that the rest of the code is being executed. Thus the download and the remaining script will *not* be synchronized; they will be "asynchronous".

```js
console.log('About to send request'); //statement 1

//send request for data to the url
fetch(url);  //statement 2

console.log('Sent request'); //statement 3
```

```
//The data is actually received sometime later,
//when the JS interpreter is down here!
```

- In the above example, the JS interpreter will execute statement 1, then statement 2 (the `fetch()` call). It will then precede *immediately* to statement 3 (the second `console.log`), without waiting for the request to finish! The download will continue to occur in the background, and will finish at some point later in the program—though we don't know exactly when.
- It is best to think of `fetch()` as a function that will just "*start* to download data", not one that actually downloads data!

Because `fetch()` is an **asynchronous function** (it's code is run asynchronously), it returns what is called a **Promise**. A Promise is object that holds a value which may not be available yet—you can think of a Promise as like a placeholder where the result of the asynchronous function call will eventually be stored (it is a "promise" to eventually have some data, though that promise may be kept or broken!).

- Promises are the modern way of handling asynchronous functions, but as part of the ES6 standard they are not yet available to all browsers (specifically: Internet Explorer). So you'll need to include *another polyfill* to support IE. This is also available from a CDN.

Promises have three possible states: *pending* (the data is downloading), *fulfilled* (the data has finished downloading), or *rejected* (the data failed to download and the promise was "broken"). We are able to specify *callback functions* (similar to event listeners) that occur when a pending Promise is either successfully fulfilled or has been rejected. The "on success" callback function is specified by calling the **`.then()`** function on the Promise object, and passing the "on success callback" as a parameter:

```
function onSuccessCallback(response) { //what to do when we get the response
    console.log(response);
}

//When fulfilled, execute the callback function (which will be passed the response)
let promise = fetch(url);
promise.then(onSuccessCallback);

//It is much more common to use anonymous variables/callbacks:
fetch(url).then(function(response) {
    console.log(response);
});
```

The "on success" callback will be passed a single parameter: the **data value** that the Promise was made for (e.g., the data that will eventually be downloaded from `fetch()`). So when the callback is executed, you will have access to the data! For example, when the `fetch()` Promise is fulfilled, it will pass an object representing the *response* to the HTTP Request:

```
let promise = fetch(url);
promise.then(function(response){
    console.log( response.url ); //a string of where the request was sent
    console.log( response.status ); //the HTTP status code (e.g., 200, 404)
});
```

This response object does have a `body` property that represents the "body" (data content) of the HTTP response. However, that body stored as a "stream" of 0s and 1s, not as a JavaScript object (or even a string you can `JSON.parse()`)! In order to get the body into a format you can use, you will need to "encode" it into a JavaScript object by calling the **`.json()`** method on it.

- There is also an equivalent `.text()` method to encode a response body into plain text.

Chaining Promises

But there's a catch: the "encoding" process performed by the `.json()` might take some time (particularly for a large amount of data). So instead of blocking (pausing) the rest of your program while that encoding occurs, the `.json()` method returns *another Promise* as a placeholder for when the encoded body is available! So you will then need to specify a `.then()` callback for *that* Promise as well.

However, a Promise's `.then()` function has a neat property that makes this easy to do. Calling the `.then()` function on a Promise returns a new Promise as a placeholder for any data produced by the `.then()` function. This promised data will be whatever value is *returned* by the "on success" callback function. This allows you to in effect "chain" `.then()` calls together, each of which can perform some kind of transformation on the data:

```
function makeQuestion(dataString) { //a function to make a string a question
    return dataString + '???';
}

//image a hypothetical asynchronous function `getAsyncString`
//it returns a Promise (placeholder) for a string load from a given source
let originalPromise = getAsyncString(myDataSource);

//when the original promise is fulfilled, call `makeQuestion` on it
//`questionPromise` will be a placeholder for that transformed data
let questionPromise = originalPromise.then(makeUpperCase);
```

```
//when the `questionPromise` is fulfilled, call an anonymous callback on it
//the callback will be passed the transformed ("question") data
questionPromise.then(function(data){
    console.log(data); //data will be a question!
})
```

More commonly, we use *anonymous variables* for subsequent promises, allowing you to chain them together in a way that almost reads like English!

```
getAsyncString()
    .then(makeQuestion)
    .then(function(data){
        console.log(data);
    });
```

But wait there's more! `.then()` also has a special feature where if the "on success" callback function returns a *Promise* (rather than another kind of value), then the "outer" promise will take on the state of that new returned Promise. This means that you can just *return* a Promise from inside a `.then()` callback, and that Promise will be the subject of the subsequent `.then()` call:

```
let outerPromise = getAsyncString(myFirstSource).then(function(firstData){
    //do something with `firstData`

    let newPromise = getAsyncString(mySecondSource); //a second async call!
    return newPromise; //return the promise.
}); //`outerPromise` now takes on the state and data of `newPromise`

outerPromise.then(function(secondData){
    //do something with `secondData`, the data downloaded from `mySecondSource`
});
```

Going back to `fetch()` to bring it all together: since the `.json()` encoding function returns a Promise, you can simply *return* that Promise from the `.then()` callback in order to make it available to subsequent `.then()` calls!

```
fetch(url)  //start the download
    .then(function(response) {  //when done downloading
        let dataPromise = response.json();  //start encoding into an object
        return dataPromise;  //hand this Promise up
    })
    .then(function(data) {  //when done encoding
        //do something with the data!!
        console.log(data); //will now be encoded as a JavaScript object!
    });
```

This code example will allow you to download data and encode it into a plain old JavaScript object that you can work with.

Handling Errors

When downloading data from the internet, it is always possible that the HTTP request may fail. The request may be sent to the wrong URL, the client computer may be having connection problems, or the receiving server may be having problems.

In order to deal with inevitable errors, Promises provide a **`.catch()`** method that is used to specify a callback that should occur if the Promise is *rejected* (an error occurs). This callback function will be passed an Error object that contains details about the error.

```
fetch(url)
    .then(function(response) {  //when done downloading
        return response.json();  //second promise is anonymous
    })
    .then(function(data) {  //when done encoding
        //do something with the data!!
        console.log(data); //will now be encoded as a JavaScript object!
    })
    .catch(function(err) {
        //do something with the error
        console.error(err);  //e.g., show in the console
    });
```

- Importantly, the `.catch()` method will "catch" errors from *all previous* Promises in a `.then()` chain! This means that the above `.catch()` will show both errors in the downloading (`.fetch()`), and errors in the body encoding (`.json()`).
- You will almost always want to show the error to the user in some way, such as by creating an alert element in the DOM.
- The `.catch()` function *also* returns a Promise, so you can continue to chain `.then()` calls after it. These later callbacks will only be executed if no previous Promise has been rejected (that is, there haven't been any errors yet).

Important: a Promise will only be rejected if there is an actual "Error" in sending the request. If the server replies with a 401 error (e.g., you didn't have permission to access the resource) or just the message "invalid API key", that won't be handled by `.catch()`. From JavaScript's perspective, the request went through perfectly—it's not fetch's fault that the data you asked for wasn't what you actually wanted!

- You can use the `response.status` and `response.ok` properties to check the status of the HTTP response.

As such, you will want to make sure to handle things like bad responses or unexpected response bodies, both in testing your application (to make sure the request is sent to the correct URL) and when handling any user input.

Resources

- An Introduction to AJAX for Front-End Developers (tuts+)
- An Introduction to `fetch()` (Google)
- JavaScript Promises: an Introduction (Google)

Chapter 15

ES6 Features

As discussed in Chapter 10, the ECMAScript specification for the JavaScript language has gone through several different versions, each of which adds new syntax and features to try and make the language more powerful or easier to work with. In 2015, a new version of the language was released—officially called "ECMAScript 2015", most developers refer to it by the working name **"ES6"** (e.g., version 6 of the language).

This chapter introduces some of the most notable and useful features introduced in ES6—particularly those that will be utilized by the React framework (discussed in the following chapters).

ES6 is mostly supported by modern browsers, with the notable exception of Internet Explorer. However, the JavaScript interpreter in older browsers (and IE) won't be ale to recognize the new syntax introduced in this version. Instead, you would need to covert that code into equivalent ES5 (or earlier) code, which can be understood. The easiest way to do this is with the Babel compiler, which will "*transpile*" JavaScript code from one version to another. The next chapter discusses how to perform this transpiling with React (spoiler: it's automatically performed by provided build tools), but it is also possible to install and utilize the Babel compiler yourself.

15.1 Classes

While JavaScript is primarily a *scripting* and functional language, it does support a form of **Object Oriented Programming** like that used in the Java language. That is, we are able to define **classes** of data and methods that act on that data, and then **instantiate** those classes into **objects** that can be manipulated. ES6 introduces a new `class` syntax so that creating classes in JavaScript even *looks* like how you make classes in Java!

Why Classes?

The whole point of using classes in programming—whether Java or JavaScript—is to perform **abstraction**: we want to be able to *encapsulate* ("group") together parts of our code so we can talk about it at a higher level. So rather than needing to think about the program purely in terms of `Numbers`, `Strings`, and `Arrays`, we can think about it in terms of `Dogs`, `Cats` or `Persons`.

In particular, classes *encapsulate* two things:

1. The **data** (variables) that describe the thing. These are known as *attributes*, *fields* or *instance variables* (variables that belong to a particular *instance*, or example, of the class). For example, we might talk about a `Person` object's `name` (a String), `age` (a Number), and Halloween haul (an array of candy).

2. The **behaviors** (functions) that operate on, utilize, or change that data. These are known as *methods* (technically *instance methods*, since they operate on a particular instance of the class). For example, a `Person` may be able to `sayHello()`, `trickOrTreat()`, or `eatCandy()`.

In JavaScript, an Object's properties can be seen as the *attributes* of that object. For example:

```
let person = {
   name: 'Ada',
   age: 21,
   costume: 'Cheshire Cat'
   trickOrTreat: function(newCandy){
      this.candy.push(newCandy);
   }
}

//tell me about this person!
console.log(person.name + " is a " + person.costume);
```

This Object represents a thing with `name`, `age` and `costume` attributes (but we haven't yet indicated that this Object has the *class*ification of "Person"). The value of the `trickOrTreat()` property is a function (remember: functions are values!), and is an example of how an Object can "have" a function.

- JavaScript even uses the `this` keyword to refer to *which* object that function being called on, just like Java! See below for more on the `this` keyword and its quirks.

A **Class** (*class*ification) acts as *template/recipe/blueprint* for individual objects. It defines what data (attributes) and behaviors (methods) they have. An object is an "instance of" (example of) a class: we **instantiate** an object from a class. This lets you easily create multiple objects, each of which can track and modify its own data. ES6 classes provide a syntax by which these “templates” can be defined.

Review: Classes in Java

First, consider the following simple class defined in *Java* (which should be review from earlier programming courses):

```java
//class declaration
public class Person {

    //attributes (private)
    private String firstName;
    private int age;

    //a Constructor method
    //this is called when the class is instantiated (with `new`)
    //and has the job of initializing the attributes
    public Person(String newName, int newAge){
        //assign parameters to the attributes
        this.firstName = newName;
        this.age = newAge;
    }

    //return this Person's name
    public String getName() {
        return this.firstName; //return own attribute
    }

    //grow a year
    public void haveBirthday() {
        this.age++; //increase this person's age (accessing own attribute)
    }

    //a method that takes in a Person type as a parameter
    public void sayHello(Person otherPerson) {
        //call method on parameter object for printing
        System.out.println("Hello, " + otherPerson.getName());

        //access own attribute for printing
        System.out.println("I am " + this.age +  " years old");
    }
}
```

You can of course utilize this class (instantiate it and call its methods) as follows:

```
public static void main(String[] args) {
    //instantiate two new People objects
    Person aliceObj = new Person("Alice", 21);
    Person bobObj = new Person("Bob", 42);

    //call method on Alice (changing her fields)
    aliceObj.haveBirthday();

    //call the method ON Alice, and PASS Bob as a param to it
    aliceObj.sayHello(bobObj);
}
```

A few things to note about this syntax:

1. You declare (announce) that you're defining a class by using the `class` keyword.
2. Java attributes are *declared* at the top of the class block (but assigned in the constructor).
3. Classes have **constructor** methods that are used to instantiate the attributes.
4. Class methods are declared *inside* the class declaration (inside the block, indenting one step).
5. Class methods can access (use) the object's attribute variables by referring to them as `this.attributeName`.
6. You **instantiate** objects of the class's type by using the `new` keyword and then calling a method with the name of the class (e.g., `new Person()`). That method *is* the constructor, so is passed the constructor's parameters.
7. You call methods on objects by using **dot notation** (e.g., `object.methodName()`).
8. Instantiated objects are just variables, and so can be passed into other methods.

ES6 Class Syntax

Here is how you would create *the exact same class* in JavaScript using ES6 syntax:

```
//class declaration
class Person {

    //a Constructor method
    //this is called when the class is instantiated (with `new`)
    //and has the job of initializing the attributes
    constructor(newName, newAge) {
        //assign parameters to the attributes
        this.firstName = newName;
        this.age = newAge;
    }

    //return this Person's name
    getName() {
        return this.firstName; //return own attribute
    }

    //grow a year
    haveBirthday() {
        this.age++; //increase this person's age (accessing own attribute)
    }

    //a method that takes in a Person type as a parameter
    sayHello(otherPerson) {
        //call method on parameter object for printing
        console.log("Hello, " + otherPerson.getName());

        //access own attribute for printing
        console.log("I am " + this.age +  " years old");
    }
}
```

And here is how you would use this class:

```
//instantiate two new People objects
let aliceObj = new Person("Alice", 21);
let bobObj = new Person("Bob", 42);

//call method on Alice (changing her attributes)
aliceObj.haveBirthday();

//call the method ON Alice, and PASS Bob as a param to it
aliceObj.sayHello(bobObj);
```

As you can see, this syntax is *very*, ***very*** similar to Java! Just like with JavaScript functions, most of the changes have involved getting rid of type declarations. In fact, you can write a class in Java and then just delete a few words to make it an ES6 class.

Things to notice:

1. *Just like in Java*, JavaScript classes are declared using the `class` keyword (this is what was introduced in ES6).

   Always name classes in PascalCase (starting with an Upper case letter)!

2. JavaScript classes **do not** declare attributes ahead of time (at the top of

the class). Unlike Java, JavaScript variables always "exist", they're just `undefined` until assigned, so you don't need to explicitly declare them.

- In JavaScript, nothing is private; you effectively have `public` access to all attributes and functions.

3. JavaScript classes always have only one **constructor** (if any), and the function is simply called `constructor()`.

    - That's even clearer than Java, where you only know it's a constructor because it lacks a return type.

4. *Just like in Java*, JavaScript class methods are declared *inside* the class declaration (inside the block, indenting one step).

    But note that you don't need to use the word `function` to indicate that a method is a function; just provide the name & parameters. This is because the only things *in* the class are functions, so declaring it as such would be redundant.

5. *Just like in Java*, JavaScript class methods can access (use) the object's attribute variables by referring to them as `this.attributeName`.

6. *Just like in Java*, you **instantiate** objects of the class's type by using the `new` keyword and then calling a method with the name of the class (e.g., `new Person()`). That method *is* the `constructor()`, so is passed the constructor's parameters.

7. *Just like in Java*, you call methods on objects by using **dot notation** (e.g., `object.methodName()`).

8. *Just like in Java*, instantiated objects are just variables, and so can be passed into other methods.

So really, it's just like Java—except that for the differences in how you declare functions and the fact that we use the word `constructor` to name the constructor methods.

The other difference is that while in Java we usually define each class inside it's own file, in JavaScript you often create multiple classes in a single file, at the same global "level" as you declared other, non-class functions:

```
//script.js
'use strict';

//declare a class
class Dog {
    bark() { /*...*/ }
}

//declare another class
```

```
class Cat {
    meow() { /*...*/ }
}

//declare a (non-class) function
function petAnimal(animal) { /*...*/ }

//at the "main" level, instantiate the classes and call the functions
let fido = new Dog();
petAnimal(fido); //pass this Dog object to the function
```

Although the above syntax looks like Java, it's important to remember that JavaScript class instances are still *just normal Objects like any other.* For example, you can add new properties and functions to that object, or overwrite the value of any property. Although it looks like a Java class, it doesn't really behave like one.

Inheritance

The ES6 `class` syntax also allows you to specify **class inheritance**, by which one class can `extend` another. Inheritance allows you to specify that one class is a *more specialized version* of another: that is, a version of that class with "extra abilities" (such as additional methods).

As in Java, you use the `extends` keyword to indicate that one class should **inherit** from another:

```
//The "parent/super" class
class Dog {
  constructor(name) {
      this.name = name;
  }

  sit() {
      console.log('The dog '+this.name+' sits. Good boy.');
  }

  bark() {
      console.log('"Woof!"');
  }
}

//The "child/sub" class (inherits abilities from Dog)
class Husky extends Dog {
    constructor(name, distance) {
        super(name); //call parent constructor
        this.distance = distance;
    }

    //a new method ("special ability")
    throwFootball() {
        console.log('Husky '+this.name+' throws '+this.dist+' yards');
    }

    //override (replace) parent's method
    bark() {
      super.bark(); //call parent method
      console.log("(Go huskies!)");
    }
}

//usage
let dog = new Husky("Harry", 60); //make a Husky
dog.sit(); //call inherited method
dog.throwFootball(); //call own method
dog.bark(); //call own (overridden) method
```

In this case, the class `Husky` is a *specialized version* of the class `Dog`: it ***is a*** `Dog` that has a few special abilities (e.g., it can throw a football). We refer to the base, less specialized class (`Dog`) as the **parent** or **super class**, and the derived, more specialized class (`Husky`) as the **child** or **sub-class**.

The sub-class `Husky` class **inherits** the methods defined in its parent: even though the `Husky` class didn't define a `sit()` method, it still has that method define because the *parent* has that method defined! By extending an existing class, you get can get a lot of methods for free!

The `Husky` class is also able to **override** its parents methods, defining it's own specialized version (e.g., `bark()`). This is useful for adding customization, or for providing specific implementations of callbacks that may be utilized by a framework—a pattern that you'll see in React.

Note that despite this discussion, *JavaScript is not actually an object-oriented language*. JavaScript instead uses a prototype system for defining types of Objects, which allows what is called prototypical inheritance. The ES6 `class` keyword doesn't change that: instead, it is simply a "shortcut syntax" for specifying Object prototypes in the same way that has been supported since the first version of JavaScript. The `class` keyword makes it easy to define something that looks and acts like an OOP class, but JavaScript isn't object-oriented! See this (detailed) explanation for further discussion.

15.2 Arrow Functions

As described in Chapter 11, JavaScript lets you define functions as *anonymous values*:

```
let sayHello = function(name) {
  return 'Hello '+name;
}
```

As you have seen, the use of anonymous functions is incredibly common in JavaScript, particularly when used as anonymous callbacks. Because this is so common, ES6 introduced a simpler, more concise *shortcut syntax* for quickly specifying anonymous functions. Though officially called lambda functions, they are more commonly known as **arrow functions**, because of how they utilize an "arrow" symbol `=>`:

```
let sayHello = (name) => {
    return 'Hello '+name;
}
```

To turn an anonymous function into an *arrow function*, you just remove the `function` keyword from the definition, and place an arrow `=>` between the parameter list and the block (indicating that the parameter list "goes to" the following block). This saves you a couple of characters when typing.

And for simple callback functions, you can make this even *more* compact:

1. If the function takes only a single parameter, you can leave off the `()` around the parameter list.
2. If the function body has only a single statement, you can leave the `{}` off the block.
3. If the function body has only a single statement *AND* that statement returns a value, you can leave off the `return` keyword (the "single statement" arrow function returns the result of the last statement, which will either be an expression or `undefined`).

Thus the above `sayHello` method could be written using **concise body** format as:

```
let sayHello = name => 'Hello '+name;
```

- I recommend you always leave the parentheses `()` on the parameter list, as it helps with readability (and makes it easier to adjust the parameters later)!

Note that if a function takes no parameters, you **must** include the `()` to indicate it is an arrow function:

```
//normal function syntax
let printHello = function() {
    console.log('Hello world');
}

//arrow syntax
let printHello = () => {
    console.log('Hello world');
}

//concise body
let printHello = () => console.log('Hello world');
```

- Despite the above example, it's better style to include the `{}` around a function body that doesn't return a value.

Arrow functions are particularly useful for specifying *anonymous callback functions*:

```
let array = [1,2,3]; //an array to work with

//normal function
array.map(function(num) {
  return num*2; //multiply each item by 2
});

//arrow syntax
array.map(num => {
  return num*2; //multiply each item by 2
});

//concise body
array.map(num => num*2);
```

Arrow functions are great and you should always use them for anonymous callback functions (if your target platform suppo them). They have quickly become the standard way of writing JavaScript, and thus you will see them all over examples and professionally written code.

Working with this

In JavaScript, functions are called on **Objects** by using *dot notation* (e.g., `myObject.myFunction()`). Inside a function, you can refer to the Object that the function was called on by using the **`this`** keyword. `this` is a local variable that is *implicitly assigned* the Object as a value.

```
let doggy = {
  name: "Fido",
  bark: function() {
      console.log(this.name, "woofs"); //`this` is object the function was called on
  }
}

doggy.bark(); //=> "Fido woofs"
```

- Here the `this` is assigned the object `doggy`, what `.bark()` was called on.

But because functions are values and so can be assigned to multiple variables (given multiple labels), the object that a function is called on may not necessarily be the object that it was first assigned to as a property. `this` refers to object the function is called on at execution time, not at the time of definition:

```
//An object representing a Dog
let doggy = {
  name: "Fido",
  bark: function() { console.log(this.name + " woofs"); }
}

// An object representing another Dog
let doggo = {
  name: "Spot",
  bark: function() { console.log(this.name + " yips")}
}

//This is Fido barking
doggy.bark( /*this = doggy*/ ); //=> "Fido woofs"

//This is Spot barking
doggo.bark( /*this = doggo*/ ); //=> "Spot yips"

//This is Fido using Spot's bark!
doggy.bark = doggo.bark; //assign the function value to `doggy`
doggy.bark( /*this = doggy*/) //=> "Fido yips"
```

- Notice how the `this` variable is implicitly assigned a value of whatever object it was called on—even the function is assigned to a new object later!

But because the `this` variable refers to the object the function is called on, problems can arise for *anonymous callback functions* that are not called on any object in particular:

```
class Person {
   constructor(name){ this.name = name; } //basic constructor

   //greet each person in the given array
   greetAll(peopleArray) {
      //loop through each Person using a callback
      peopleArray.forEach(function(person) {
         console.log("Hi"+person.name+", I'm "+this.name);
      });
   }
}
```

In this example, the `greetAll()` function will produce an error: `TypeError: Cannot read property 'name' of undefined`. That is because the `this` is being called from within an anonymous callback function (the `function(person){...}`)—and that callback isn't being called on any particular object (notice the lack of dot notation). Since the anonymous callback isn't being executed on an object, `this` is assigned a value of `undefined` (and you can't access `undefined.name`).

The solution to this problem is to use *arrow functions*. An arrow function has the special feature that it shares the same lexical `this` as its surrounded code: that is, the `this` will not be reassigned to a (non-existent) object when used within an arrow function:

```
class Person {
   constructor(name){ this.name = name; }

   greetAll(peopleArray) {
      peopleArray.forEach((person) => { //arrow function (subtle difference)
         console.log("Hi"+person.name+", I'm "+this.name); //works correctly!
      });
   }
}
```

This property makes arrow functions invaluable when specifying callback functions, particularly once classes and objects are involved. *Always use arrow functions for anonymous callbacks!*

Alternatively, it is possible to "permanently" associate a particular `this` value with a function, no matter what object that function is called on. This is called **binding** the `this`, and is done by calling the `.bind()` method on the function and passing in the value you want to be assigned to `this`. The `.bind()` method will return a *new function* that has the value bound to it; often you will then take this new function and "re-assign" it to the old function variable:

```
//re-assign function
myFunction = myFunction.bind(thisValue);
```

This is a common pattern in React (and has some minuscule performance benefits), but for this class you should stick with arrow functions for cleanliness and readability.

15.3 Modules

So far, you've mostly be writing all of your JavaScript code in a single script (e.g., `index.js`), even if you'd included some other libraries via additional `<script>` tags. But as applications get larger and more complex, this single script file can quickly become unwieldy with hundreds or thousands of lines of code implementing dozens of features. Such large files become difficult to read and debug ("where *was* that function defied?"), can introduce problems with the shared global namespace ("did I already declare a `user` variable?"), and overall mixes code in a way that violates the *Separation of Concerns* principle.

The solution to this problem is to split up your application's code into separate **modules** (scripts), each of which is responsible for a separate piece of functionality. And rather than loading each module through a separate `<script>` tag (potentially leading to ordering and dependency issues while continuing to pollute the global namespace), you can define each module as a self-contained script that explicitly "imports" (loads) the functions and variables it needs from other modules. This allows you to better organize your program as it gets large.

While separating code into modules is a common in the Node.js environment, ES6 adds syntax that allows you to treat individual `.js` files as modules that can communicate with one another. These are known as **ES6 Modules**.

Browsers do not yet support this syntax! However, modern applications are usually packaged using a *bundling* build tool such as webpack, which will do the work of "compiling" modules into a single script file (and that we will use with React).

Note that the ECMAScript committee is currently a specification for browser-native module loaders. For example, it is possible to try out modules in the latest versions of Chrome (as of **September 2017**) and Safari (as of March 2017) by specifying the `type=module` attribute for a `<script>` tag (and loading the page via a web server). But the most common solution will be to use an external loading app like webpack.

As in Java, ES6 Modules are able to "load" external modules or libraries by using the **`import`** keyword:

```
//Java example: import `Random` variable from `java.util` to use globally
import java.util.Random
```

```
//JavaScript: import the `Random` variable from a `util.js` module
import { Random } from './util';
```

This is most common version of the ES6 `import` syntax: you write the keyword `import`, following by a set of braces `{ }` containing a comma-separated sequence of variables you wish to "import" from a particular module. This is followed by the `from` keyword, followed by a string containing the **relative path** to the module script to import. Note that including the extension of the module script is usually optional (by default, ES6 will look for files ending in `.js`).

- Be sure to include the `./` to indicate that you're loading a file from a particular directory. If you leave that off, the module loader will look for a module installed on the *load path* (e.g., in `node_modules/`). That is how you load external files such as jQuery though:

  ```
  //with jquery installed in `node_modules/`
  //import the `$` and `jQuery` variables
  import {$, jQuery} from 'jquery';
  ```

If you want to make a variable available from one module to use in another, you will need to **export** it (so that it can be "imported" elsewhere). You do this by using the **`export`** keyword, placed in front of the variable declaration:

```
/*** my-module.js ***/
export let question = "Why'd the chicken cross the road?";
export let answer = "To get to the other side";
export function laugh() {
    console.log("hahaha");
}
```

```
/*** index.js ***/
import { question, answer, laugh } from './my-module';

console.log(question); //=> "Why'd the chicken cross the road?"
console.log(answer); //=> "To get to the other side"
laugh(); //"hahaha"
```

- Note that you can export functions as well as variables, since functions *are* values!
- Once a value has been imported, it is available globally (just as if it were defined in a previous `<script>` tag)

There are a number of ways to export and import variables, giving you significant customization on which values you wish to share and load:

```
/*** my-module.js ***/
export function foo() { return 'foo'; } //make available (as above)

function bar() { return 'bar'; }
export bar; //export previously defined variable

export { bar as barFunction }; //provide an "alias" (consumer name) for value

//will not be available (a "private" function)
function baz() { return 'baz'; }
```

```
/*** index.js ***/
import {foo, barFunction} from './my-module'; //import multiple values
foo() //=> 'foo'
barFunction() //=> 'bar'

import {foo as myFoo} from './my-module'; //provide an "alias" for value
myFoo(); //=> 'foo'

import * as theModule from './my-module'; //import everything that was exported
                                          //loads as a single object with values
                                          //as properties
theModule.foo(); //=> 'foo'
theModule.barFunction(); //=> 'bar'
theModule.baz(); //Error [private function]
```

Note the additional syntax options in this example:

- You can use the **`as`** keyword to “alias” a variable either when it is exported (so it is shared with a different name) or when it is imported (so it is loaded and assigned a different name). This is particularly useful when trying to produce a clean API for a module (so you `export` values with consistent names, even if you don’t use those internally), or when you want to `import` a variable with a very long name.

- It is possible to just `import` *everything* that was exported by a module using the **`import * as`** syntax. You specify an object name that will represent the values exported module (e.g., `theModule` in the example), and each exported variable will be a *property* of that object. This is particularly useful when you may be adding extra `exports` to a module during development, but don’t want to keep adjusting the `import` statement.

- Note also that *only* the variables you `export` are made available to other modules! This allows you to make variables and functions that are “private” to a module!

Finally, each module can also `export` a *single* (just one!) **default** variable, which provides a slight shortcut when importing the variable from that module. You specify the *default export* by including the **`default`** keyword immediately after `export`:

```
/*** my-module.js ***/
export default function sayHello() {
    return 'Hello world!';
}
```

```
/*** index.js ***/
import greet from './my-module';
```

```
greet(); //=> "Hello world!"
```

- When importing a `default` export, you just provide the variable name ("alias") you wish to refer to that exported value by.
- Note that it is also possible to make anonymous values into `default` exports:

  ```
  /*** animals.js ***/
  export default ['lion', 'tiger', 'bear']; //export anonymous array
  ```

The `default` export technique is particularly common in object-oriented frameworks like React, where you can make each JavaScript module contain the code for a single `class`. That single `class` can be made the `default` export, allowing other modules to import it quickly and easily as `import MyComponent from './MyComponent.js'`.

This section describes ES6 modules, which are used with web applications (being part of the ECMAScript specification). However, Node.js utilizes an alternate module loading system called CommonJS. This uses the built-in method `require()` to load a module (which returns a single "exported" variable as a result). Values are exported from a module by assigning them to the `module.exports` global. This coure will exclusively utilize ES6 Modules, but it's good to be aware of the alternate CommonJS approach when searching for help.

15.4 Other Features

> Syntactic Sugar causes cancer of the semicolon - Alan Perlis

There are a few other notable syntax options provided by ES6 that you will likely come across.

Template Strings

In ES6, you can declare Strings that contain embedded expressions, allowing you to "inject" an expression directly into a string (rather than needing to concatenate the String with that expression). These are known as **template strings** (or *template literals*). Template strings are written in back ticks (**``**) rather than quotes, with the injected expressions written inside of a **${}** token:

```
let name = 'world';
let greeting = `Hello, ${name}!`; //template string
console.log(greeting); //=> "Hello, world!"
```

- Template strings can also include line breaks, allowing you to make multi-line strings!

Note that you can put *any* expression inside the `${}` token; however, it's best practice to keep the expression as simple as possible (such as by using a local variable) to support readability:

```
let name = 'world';

//greeting with capitalization. Don't do this!
let greeting = `Hello, ${name.substr(0,1).toUpperCase() + name.substr(1)}!`
console.log(greeting); //=> "Hello, world!";

//do this instead!
let capitalizedName = name.substr(0,1).toUpperCase() + name.substr(1);
let greeting = `Hello, ${capitalizedName}`
console.log(greeting); //=> "Hello, world!";
```

Destructuring and Spreading

ES6 also introduced **destructing assignments**, which allow you to assign each element of an array (or each property of an object) into separate variables all in a single operation. You do this by writing the variables you wish to assign to inside of `[]` on the *left-hand side* of the assignment—almost like you are assigning to an array!

```
let array = [1, 2, 3]
let [x, y, z] = array; //assign array elements to `x`, `y`, `z` respectively
console.log(x); //=> 1;
console.log(y); //=> 2;
console.log(z); //=> 3;
```

If the array contains more than the target number of elements the extra elements will be ignored. However, you can capture them by using the **spread operator** (`...`), which is used to either gather or spread a sequence of values (e.g., a list of parameters) into or across an array:

```
let dimensions = [10, 20, 30, 40];
let [width, height, ...rest] = dimensions
console.log(width);  //=> 10
console.log(height); //=> 20
console.log(rest);   //=> [30, 40];
```

More commonly, the *spread operator* is used to specify that a function can take an undefined number of arguments, and to gather all of these objects into a single array:

```
//a function that logs out all of the parameters
function gather(...args){
    //all the parameters will be grouped into a single array `args`
    args.forEach((arg) => {
        console.log(arg) //can log out all of them, no matter how many!
    });
}

gather('a', 'b', 'c'); //=> "a" "b" "c"
gather(1,2,3,4,5,6); //=> "1" "2" "3" "4" "5" "6"

//a function that adds up all the arguments (no matter how many)
function sum(...numbers) {
    //number is an array, so we can `reduce()` it!
    let total = numbers.reduce((runningTotal, num) => {
        return runningTotal + num; //new total
    }, 0); //start at 0

    return total;

    //or as one line with a concise arrow function:
    return numbers.reduce((total, n) => total+n);
}

sum(3,4,3); // => 10
sum(10,20,30,40); // => 100
```

These are a few of the more common and potentially useful ES6 features. However, remember that most of these are just "syntactic shortcuts" for behaviors and functionality you can already achieve using ES5-style JavaScript. Thus you don't need to utilize these features in your code (though they can be helpful), and they will often show up in how we utilize libraries such as React.

Arrow functions are not optional. **Always use arrow functions for anonymous callbacks!**

Resources

- ES6 For Humans an good online text on ES6
- ECMAScript 6 A good unofficial summary with examples
- ES6 Features (also shows equivalent ES5 syntax where possible)
- ECMAScript 2015 Language Specification official standard

Chapter 16

Introduction to React

This chapter introduces the **React** JavaScript library. React is "a JavaScript library for building user interfaces" developed by Facebook (though it is released as open-source software). At its core, React allows you to dynamically generate and interact with the DOM, similar to what you might do with jQuery. However, React was created to make it much easier to define and manipulate lots of different parts of the DOM, and to do so *quickly* (in terms of computer speed). It does this by enabling you to declare a web app in terms of different **components** (think: "things" on a web page) that can be independently managed—this lets you design and implement web apps at a higher level of abstraction. Moreover, components are usually associated with some set of data, and React will automatically "re-render" (show) the updated component when the data changes—and to do so in a computationally efficient manner.

React is currently the most popular "framework" for building large-scale web applications (its chief competitors being Angular and Vue.js, though there are scores of similar frameworks that are also used. Check out TodoMVC for an example of the same application in each!). This means that React is generally very well documented; there are hundreds of tutorials, videos, and examples for building React applications (of which this chapter will be yet another). Some general resources are included at the end of the chapter, but in general we recommend you start with the official documentation, particularly the set of main concepts. Note that this book more or less follows the approach used by Facebooks's Intro to React Tutorial.

React has gone through a couple of different major versions in its short life. Moreover, the "style" in which React is written has also evolved over time as new features are added to both the React library and the JavaScript language. This book introduces and emphasizes an approach that prioritizes clarity of concepts and readability of code, rather than conciseness of syntax or use of advanced options.

16.1 Getting Set Up: Create React App

React is a JavaScript library similar to those discussed in previous chapters—you load the library and then can call upon its methods. However, React makes use of a significant amount of advanced and custom JavaScript syntax (described below), including extensive use of ES6 Modules. Thus in practice, using developing React apps requires using a large number of different development tools—just getting setup and started is one of major hurdles in learning React!

Luckily, Facebook does provide an amazing application which combines many of these these different development tools together. **Create React App (CRA)** is a *command line* application that generates scaffolding ("starter code") for a React application, as well as provides built-in scripts to run, test, and deploy your code.

You can use Create React App to create a new React app by running the program and specifying a name for the folder you want the app to be created inside of. For example, the below command will create a new folder called `my-app` that contains all of the code and tools necessary to create a React app:

```
# Create a new React app project in a new `my-app` directory
# This may take a minute...
npx create-react-app my-app

# Change into new project directory to run further commands
cd my-app
```

- The `npx` command is installed alongside `npm 5.2.0` and later, and can be used to (temporarily) install and run a global command in a single step. So rather than needing to use `npm install -g create-react-app`, you can just use `npx`.
- If you specify the current directory (`.`) as the target folder name, Create React App will create a new react app in the current folder! This is useful when you want to create a React app inside an existing cloned GitHub repository.

16.1.1 Running the Development Server

After you create a React app, you can use one of CRA's provides scripts (the `start` script) to automatically start up a *development webserver*:

```
# Make sure you are in the project directory
cd path/to/project

# Run the development server script
npm start
```

This will launch the webserver, and open up a new browser window showing the rendered web page! This development webserver will automatically perform the following:

1. It will automatically *transpile* React code into pure JavaScript that can be run in the web browser (see *JSX* below).
2. It will combine (**bundle**) all of the different code modules into a single file, and automatically inject that single file into the HTML.
3. It will display *errors and warnings* in your browser window—most errors will show up in the developer console, while fatal errors (e.g., syntax errors that crash the whole app) will be shown as an error web page. Note that the Chrome React Developer Tools will provide further error and debugging options.
4. It will automatically reload the page whenever you change the source code!

All these capabilities are provided by a tool called **webpack**. Webpack is at its core a *module bundler*: it takes your complicated source code structure (lots of files) and transforms it into a brand new, "simplified" version (a few files). In the process, can automatically process and change that code—from automatic formatting and styling, to compiling or transpiling from one language to another. Webpack is one of the more complex build tools; however, Create React App provides its own configuration to this program so that you don't have to do anything with it directly!

In order to stop the server, hit `ctrl-c` on the command line.

Important: There is currently an issue with Create React App on Windows using Git Bash where hitting `ctrl-c` won't actually stop the server—if you try to run it again, you'll get an error about the port still being occupied. Current work-arounds are to use the Task Manager to end the `node` process, or to instead use the Linux subsystem for Windows.

16.1.2 Project Structure

A newly created React app will contain a number of different files, including the following:

- The `public/index.html` file is the home page for your application; when webpack builds the app. If you look at this file, you'll notice that it doesn't include any CSS `<link>` or JavaScript `<script>` elements—the CSS and JavaScript for your React app will be *automatically injected* into the HTML file when the app is built and run through the development server. All you need to do is fill in those files (found in the `src/` folder). Similarly, all of the *content* of the page will be defined as JavaScript components—you'll be creating the entire page in JavaScript files. Thus in practice, you barely modify the `index.html` file (beyond changing the `<title>` and favicon and such).

Overall the `public/` folder is where you put assets that you want to be available to your page but aren't loaded and compiled through the JavaScript (i.e., they're not "source code"). For example, this is where you would put a `img/` folder to hold pictures to show. The `public/` folder will be the "root" of the webpage; a file found at `public/img/picture.jpg` would be referenced in your DOM as `img/picture.jpg`.

- The `src/` folder contains the source code for your React app. The app is started by the `index.js` script, but this script imports and uses other JavaScript modules (such as the `App.js` script, which defines the "App" module). In general, you'll code React apps by implementing components in one or more modules (e.g., `App.js`, `AboutPage.js`), which will then be imported and used by the `index.js` file. See the *Components* section below for details.
  - The `serviceWorker.js` script is an optional component used to improve efficient in production (e.g., when you're done developing). You can ignore it or even delete it for now.
- The `src/index.css` and `src/App.css` are both CSS files used to style the entire page and the individual "App" component respectively. Notice that the CSS files are imported from *inside the JavaScript files* (with e.g., `import './index.css'`). This is because the webpack build system that Create React App uses to run apps knows how to load CSS files, so you can "include" through the JavaScript rather than needing to link them in the HTML.

  If you want to load an external stylesheet, such as Bootstrap or Font-Awesome, you can still do that by modifying the `public/index.html` file, or by installing them as a module and importing them through the JavaScript. See the documentation for *Adding Bootstrap* for an example.
- The `README.md` contains a copy of the User Guide (which can also be found online), focused on "how to perform common tasks". the user guide is *very* thorough—if you want to do something within the CRA framework, try searching this first, as there is probably a guide and example code for you to use!

In general, you'll be working with the `.js` and `.css` files found in the `src/` folder as you develop a React app.

16.1.3 Alternatives to Create React App

While Create React App is the easiest and recommended way to get started with a React application, it's also possible to try out React just by including external

JavaScript libraries in the same way you would do with jQuery. Note that this approach is *not recommended*, especially for production systems—it will run much slower, and be harder to develop (you don't get the useful debugging features of Create React App's development server).

```
<!-- Load the React library scripts (there are two) -->
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

<!-- Load the Babel script -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.26.0/babel.min.js"></sc
```

The React library actually consists of two libraries that are loaded in the previous code snippet. The `react.development.js` script is the library used to create React components, while the `react-dom.development.js` library is the library that handles DOM manipulation. The third script that you loaded (`babel`) is used to transpile from a React specific code syntax called JSX (described below). This transpiling is the "slow" part; it's better to do it once at "build time" (through webpage via Create React App) than to do it each time the page is loaded and run.

16.2 JSX

At its core, React is simply a DOM creation and rendering library—it allows you to declare new DOM elements (e.g., make a new `<h1>`) and then add that element to the webpage. This is similar to the process you might do with the DOM or the jQuery library:

```
//Create and render a new `<h1>` element using DOM methods
let element = document.createElement('hi');
element.id = 'hello';
element.classList.add('my-class');
element.textContent = 'Hello World!';

document.getElementById('root').appendChild(element);
```

The React library provides a set of functions that do similar work:

```
//Import required functions from the React libraries
import React from 'react';
import ReactDOM from 'react-dom';

//Create a new `<h1>` element using React methods
let element = React.createElement(
   'h1',                       //html tag
   {                           //object of attributes
```

```
      id        : 'hello',
      className : 'my-class'
   }
   'Hello World!'           //content
);

//Show the element in the web page (inside #root)
ReactDOM.render(element, document.getElementById('root'));
```

React's `createElement()` lets you create an element (a React version, not a pure DOM element) by specifying the element type/tag, the content, and an object that maps from attribute names to values—all as a single statement. The **`ReactDOM.render()`** function will take a particular React element and insert it as the child of the specified DOM element (usually a `div#root` on the page). Note that `ReactDOM.render()` *replaces* the content of the specified DOM element—it's a "replace" not an "append" operation.

- Notice that the `class` attribute is specified with the **`className`** property. This is because `class` is a reserved keyword in JavaScript (it declares a class).

The `React.createElement` function is wordy and awkward to type, particularly if you'll be defining a lot of DOM elements (and in React, you define the entire application in JavaScript!). Thus React lets you create and define elements using **JSX**. JSX is a *syntax extension* to JavaScript: it provide a new way of writing the language—in particular, the ability to write bare HTML tags (finally!):

```
//Create a new `<h1>` element using JSX syntax
let element = <h1 id="hello" className="myclass">Hello World</h1>;

//Show the element in the web page (inside #root)
ReactDOM.render(element, document.getElementById('root'));
```

The value assigned to the `element` variable is *not* a String (it has no quotes around it). Instead it is a *syntactical shortcut* for calling the `React.createElement()` function—the HTML-like syntax can be automatically translated into the actual JavaScrip for the method call, allowing you to write the simpler version.

- Notice that there is still a semicolon (`;`) at the end of the first line; you're not actually writing HTML, but defining a JavaScript value using syntax that mostly *looks* like HTML!
- JSX can be used to define nested elements; you are not limited to a single element:

  ```
  //JSX can define nested elements
  let header = (
      <header className="well">
  ```

```
            <h1>Hello world!</h1>
        </header>
    );
```

It's very common to put elements on their own lines, intended as if you were writing HTML directly. If you do so, you'll need to surround the entire JSX expression in parentheses `()` to tell the compiler that the line break shouldn't be read as implying a semicolon!

- Note that similar to XML, JSX syntax requires that all elements be closed; thus you need to use a closing slash at the end of empty elements:

  ```
  //JSX elements must be closed
  let picture = <img src="my_picture.png" alt="A picture" />
  ```

Since JSX is not actually valid JavaScript (you can't normally include bare HTML), you need to "translate" it into real JavaScript so that the browser can understand and execute it. This process is called **transpiling**, and is usually performed using a *compiler* program called **Babel**. Babel is yet *another* piece of the toolchain used to create React applications; however, all the details about transpiling with Babel are automatically handled by the Create React App scaffolding. "Transpile with Babel" is one of the steps performed by webpack, which is included and run from the scripts provided by Create React App (whew!).

- Yes, chaining tools together like this is how modern web development works.

In short: you write JSX code, and when you run your app with Create React App it will be turned into code that your browser can actually understand.

16.2.1 Inline Expressions

JSX allows you to include JavaScript expressions (such as variables you want to reference) directly inside of the HTML-like syntax. You do this by putting the expression inside of braces (**`{}`**):

```
let message = "Hello world!";
let element = <h1>{message}</h1>;
```

When the element is rendered, the expression and the surrounding braces will be replaced by the value of the expression. So the above JSX would be rendered as the HTML: `<h1>Hello World</h1>`.

You can put any arbitrary expression inside of the `{}`—that is, any JavaScript code that resolves to a single value. This can include math operations, function calls, ternary expressions, or even an anonymous function values (see the next chapter for examples):

```
//Including an inline expression in JSX. The expressions can be mixed directly
//into the HTML
```

```
let element = <p>A leap year has {(365 + 1) * 24 * 60} minutes!</p>;
```

You can use inline expressions most anywhere in JSX, including as the value of an attribute. You can even declare element attributes as an object and then use the *spread operator* (`...`) to specify them as an inline expression.

```
//Define a JavaScript variable
let imgUrl = 'path/to/my_picture.png';

//Assign that variable to an element attribute
let pic = <img src={imgUrl} alt="A picture" />;

//Define an object of element attributes
let attributes = {
    src: 'path/to/other_picture.png',
    alt: 'Another picture'
};

//A DOM element that includes those attributes, among others.
//This is an advanced technique
let otherPic = <img {...attributes} className='my-class' />;
```

Importantly, note that elements defined with JSX are *also* valid JavaScript expressions! Thus you can use inline expressions to include one element inside of another. The below example also

```
//Define variable content to display. One is a string, one is a React element.
let greeting = "Good morning!";
let icon = <img src="sun.png" alt="A wide awake sun" />;

//Conditionally change values based on an `isEvening` value (defined elsewhere)
if(isEvening) {
    greeting = "Good evening!";
    icon = <img src="moon.png" alt="A dozing moon" />;
}

//Include the variables inline in another React element
//Notice how the `icon` element is included as a child of the `<header>`
let header = (
    <header>
      <h1>{greeting}</h1>
      {icon}
    </header>
);
```

If you include an *array* of React elements in an inline expression, those those elements will be interpreted as siblings of the parent element:

```
//An array of React elements. Could also produce via a loop or function
```

```
let listItems = [<li>lions</li>, <li>tigers</li>, <li>bears</li>];

//Include the array as an inline expression; each `<li>` element will be
//included as a child of the `<ul>`
let list = <ul>{listItems}</ul>;
```

The above sample would produce the HTML (formatted for readability):

```
<ul>
    <li>lions</li>
    <li>tigers</li>
    <li>bears</li>
</ul>
```

This is particularly useful when using a variable number of elements, or generating elements directly from data (see the *Props and Composition* section below).

As a final note: you *cannot* include HTML comments (`<!-- -->`) inside of JSX, as this isn't actually an element type. You also can't use inline JavaScript comments (`//`), because even multi-line JSX is still a single expression (that's why the `<header>` example above is written in parentheses). Thus in order to include a comment in JSX, you can use a *JavaScript block comment* (`/* */`) inside of an inline expression:

```
let main = (
    <main>
      { /* A comment: the main part of the page goes here... */}
    </main>
)
```

(This is complex enough that inlining comments in JSX isn't very common. It's better to instead structure your code so that your JSX expressions are short and self-explanatory—such as by organizing your content into *Components*).

16.3 React Components

When creating React applications, you use JSX to define the HTML content of your webpage directly in the JavaScript. However, writing all of the HTML out as one giant JSX expression doesn't much advantage over just putting that content in `.html` files in the first place.

Instead, the React library encourages and enables you to describe your web page in terms of *components*, instead of just HTML elements. In this context, a **component** is a "piece" of a web page. For example, in a social media application, each "status post" on a timeline might be a different component, the "like" button may be its own component, the "what are you thinking about?" form would be a component, the page navigation would be a component, and so forth. Thus you can almost think of a component as a semantic abstraction of a `<div>` element—a “chunk” of the page. But like with `<div>`s, a component can be seen as made of of *other* components—you might have a “Timeline” component that is made up of many “StatusPost” components, and each StatusPost component may include a “ReactionButton” component.

Thinking about a page in terms of components instead of HTML elements can make it much easier to design and reason about your page—you don’t need to worry about the code syntax or the specific element to use, you can just think of your page in terms of its overall layout and content—what needs to be on the page, and how that content is organized—the same way you might mock up a web page in a graphic design program. See the documentation article *Thinking in React* for an example of how to design a page in terms of components.

React enables you to *implement* your page in terms of explicit components by providing a way for you to define your own XML elements! Thus you can define a `<Timeline>` element, which has multiple `<StatusPost>` elements as children. Components will be able to be included directly in your JSX, made up of and mixed in with standard HTML elements.

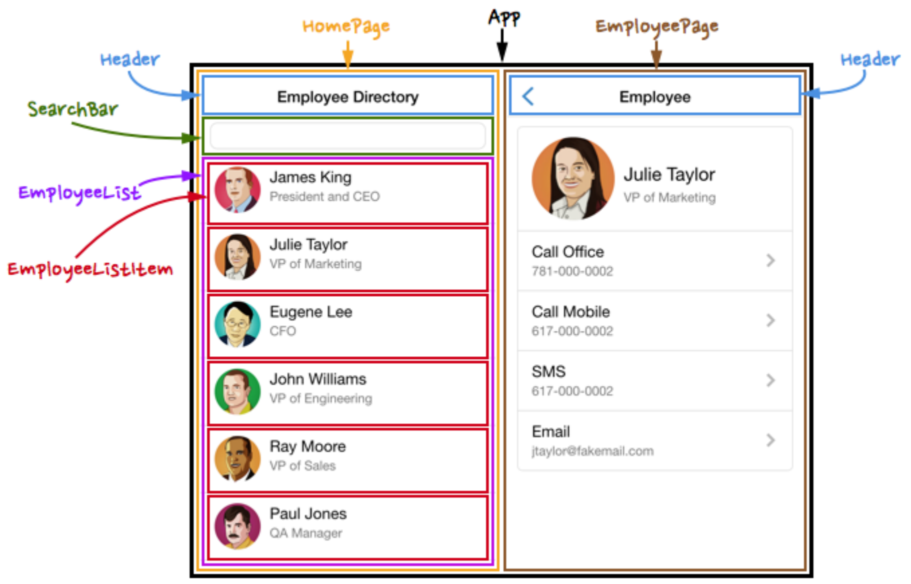


Figure 16.1: An example page broken up into components. Example from http://coenraets.org/blog/2014/12/sample-mobile-application-with-react-and-cordova/

```
<App>
  <HomePage>
    <Header />
    <SearchBar />
    <EmployeeList>
      <EmployeeListItem person="James King" />
      <EmployeeListItem person="Julie Taylor" />
      <EmployeeListItem person="Eugene Lee" />
    </EmployList>
  </HomePage>
  <EmployeePage>
    <Header />
    ...
  </EmployeePage>
</App>
```

As a component represents a "type" of content, you define a component by defining a **class** (using ES6 Class syntax) that acts as a template or blueprint for that content. In particular, you define a class that *inherits* from the **`React.Component`** class—this inheritance means that the class you define *IS A* React Component!

```
//Define a component representing information about a user
class UserInfo extends React.Component {
    //Components MUST override the `render()` function, which must
    render() {
        //This is an everyday function; you can include any code you want here
        let name = "Ethel";
        let descriptor = "Aardvark";

        //Return a React element (JSX) that is how the component will appear
        return (
            <div>
                <h1>{name}</h1>
                <p>Hello, my name is {name} and I am a {descriptor}</p>
            </div>
        )
    }
}

//instantiate the class as a new value (a React element)
let infoElement = <UserInfo />;

//Show the element in the web page (inside #root)
ReactDOM.render(infoElement, document.getElementById('root'));
```

This class defines a new component (think: a new HTML element!) called `UserInfo`. Things to note about this defining components:

- A component class **must** be named starting with a Capital letter (and usually in CamelCase). This is to distinguish a custom component from a "normal" React element. It's also proper object-oriented style to name classes with capital letters!

- A component class **must** implement a **`render()`** function. This function will be called by React (e.g., by `ReactDOM.render()`) when it needs to determine how to render a component—how that component will appear on the page. The function takes no arguments, and must return the React elements (JSX) that will be appended to the DOM.

  Note that the `render()` method can only return a *single* React element (you can't return more than one value at a time from a JavaScript function). However, that single React element can have as many children as you wish. It's very common to "wrap" all of the content you want a component to include in a single `<div>` to return, as in the above example.

  Also note that the `render()` method is a function like any other that you've written. That means that it can and often does do more than just return a single value! You can include additional code statements (variable declarations, if statements, loops, etc) in order to calculate the values you'd like to include in inline expressions in the returned JSX.

  Style requirement: perform any complex logic—including functional looping such as with `map()` or `filter()`—outside of the `return` statement. Keep the `return` statement as simple as possible to it remains readable—and more importantly *debuggable* (so you can use things like `console.log` statements to check intermediate values before they are returned). **NEVER** include an inline callback function so that you have a `return` statement inside of a `return` statement!

  Remember that you never call the `render()` method directly! Instead, this function is called by the React framework (specifically, the `RenderDOM.render()` function) when it decides that it needs to show—or re-show—the component.

- A component class can of course include other methods as well! For example, you can include "helper methods" for doing some logical work. Some specific additional methods used when managing interactions and state are described in the next chapter.

- Note that the `React.Component` class is often imported as a *named import*, so that you can refer to it directly:

  ```
  import {Component} from 'react';

  class MyComponent extends Component {
  ```

```
        render() {
            //...
        }
    }
```

Once it's been defined, that component can then be instantiated using XML-style syntax, just like any other element (i.e., as `<UserInfo />`). You can use this like any other React element, such as including it as a child of other HTML (e.g., nest it inside of a `<div>`). A component thus *encapsulates* a chunk of HTML, and you can almost think of it as a "shortcut" for including that HTML.

It's also possible to define very basic components (those that don't need to manage state) as regular JavaScript functions. These functions are passed the props as an argument, and return the React element (JSX) to render. In a way, it's like defining a class that is just a render function. However, this text will exclusively use ES6 class syntax to define components, as it makes it more clearly explicit that you are defining new classes of component (as well as being easier to later modify when you inevitably decide to track state). Neverthless, you will see other examples and libraries use function syntax for defining components—if you see a function with a Capitalized name, know that it's actually a class for a Component!

Additionally, older versions of React included a function `React.createClass()` that took in an *object* whose properties were functions that would act as class methods (see here for details). Although this function has been removed, it's worth being aware of in case you come across older examples of React code that you need to interpret or adapt.

Because components are instantiated as React elements, that means that you can use them anywhere you would use a normal HTML element—including in the `render()` function of another component! This is called *composing* components—you are specifying that one component "composed" (made up of) others:

```
//A component representing a message
class HelloMessage extends Component {
  render() {
     return <p>Hello World!</p>;
  }
}

//A component representing another message
class GoodbyeMessage extends Component {
  render() {
     return <h1>See ya later!</h1>;
  }
}
```

```
//A component that is composed of (renders) other components!
class MessageList extends Component {
  render() {
     return (
       <div>
         <h1>Messages for you</h1>
         <HelloMessage /> {/* A HelloMessage component */}
         <GoodbyeMessage /> {/* A GoodbyeMessage component */}
       </div>
     );
  }
}

//Render an instance of the "top-level" ("outer") component
ReactDOM.render(<MessageList />, document.getElementById('root'));
```

In the above example, the `MessageList` component is composed of (renders) instances of the `HelloMessage` and `GoodbyeMessage` classes; when the `MessageList`'s `render()` function is called, the interpreter will instantiate the two other components, rendering each in turn. In the end, all of the content rendered will be standard HTML elements—they've just been organized into components.

- Components are usually instantiated as empty elements, though it is possible to have them include child elements.
- Notice that you can and do mix standard HTML elements and components together; components are just providing a useful abstraction for organizing content, logic, and data!

In practice, React applications are made up of many different components—dozens or even hundreds depending on the size of the app! The "top-most" component is commonly called **`App`** (since it represents the whole "app"), with the `App`'s `render()` function instantiating further components, each of which represents a different part of the page.

See *Props and Composition* below for more examples of common ways that React components are composed.

16.3.1 Component Organization

React applications will often have dozen of components, which can quickly grow more complex than the basic starting examples in the previous section. If you tried to define all of these components in a single `index.js` file, your code would quickly become unmanageable.

Thus individual components are usually defined inside of separate **modules** (files), and then *imported* by the modules that need them&dmdash;other components or the root `index.js` file.

Recall that in order to make a variable, function, or class (e.g., a Component) available to other modules, you will need to `export` that component from its own module, and then `import` that value elsewhere:

```
/* in Messages.js file */

//Export the defined components (as named exports)
export class HelloMessage extends Component { /* ... */ }

export class GoodByeMessage extends Component { /* ... */ }
```

```
/* in App.js file */

//Import components needed from other modules
import { HelloMessage, GoodbyeMessage } from `./Messages.js`; //named imports!

//Can also export as a _default_ export; common if the file has only one component
export default class App extends Component {
    render() {
        return (
            <div>
                {/* Use the imported components */}
                <HelloMessage />
                <GoodbyeMessage />
            </div>
        )
    }
}
```

```
/* in index.js file */

//Import components needed from other modules
import App from `./App.js`; //default import!

//Render the imported component
ReactDOM.render(<App />, document.getElementById('root'));
```

This structure allows you to have each component (or set of related components) in a different file, helping to organize your code.

- Notice that this "export-import" structure implies *one-directional* hierarchy: the `Message` components don't know about the `App` component that will render them (they are self-contained). You should not have two files that import from each other!

- The `index.js` file will usually just import a single component; that file stays very short and sweet. Put all of your component definitions in other files

It is common to use the file system to further organize your component modules, such as grouping them into different folders, as in the below example:

```
src/
|-- components/
  |-- App.js
  |-- navigation/
    |-- NavBar.js
  |-- utils/
    |-- Alerts.js
    |-- Buttons.js
index.js
```

Note that a module such as `Alerts.js` might define a number of different "Alert" components to use; these could be imported into the `NavBar.js` or `App.js` modules as needed. The `App.js` component would be imported by the `index.js` file, which would contain the call to `ReactDOM.render()`.

Note that it is not necessary to put each different component in a separate module; think about what kind of file organize will make it easy to find your code and help others to understand how your code is structured!

16.4 Properties (`Props`)

One of the main goals of using a library such as React is to support *data-driven views*—that is, the content that is rendered on the page can be based on some underlying data, providing a visual representation of that data. And if that underlying data changes, React can automatically and efficiently "re-render" the view, making the page dynamic.

We can specify the underlying data that a React component will be representing by specifying **properties** (usually just called **props**) for that component. Props are the "input parameters" to a component—they are the details you *pass* to it so that it knows what and how to render its content.

You pass a prop to a component when you instantiate it by specifying them as XML attributes (using `name=value` syntax)—the props are the "attributes" of the component!

```
//Passing a prop called `message` with value "Hello property"
let messageA = <MessageItem message="Hello property!" />;

//Passing a prop using an inline expression
```

```
let secret = "Shave and a haircut";
let messageB = <MessageItem message={secret} />;

//A component can accept multiple props
//This component takes in a `userName` property as well as a `descriptor` property
let userInfo = <UserInfo userName="Ethel" descriptor="Aardvark" />;
```

- Prop names must be valid JavaScript identifiers, and so should be written in camelCase (like variable names).

Components are usually defined so that they expect a certain set of props—similar to how a function is defined to expect a certain set of arguments. It is possible to pass "extra" props to the Component (by specifying additional XML attributes), but if the component doesn't expect those props it won't do anything with them. Similarly, failing to pass an expected prop will likely result in errors—just like if you didn't pass a function an expected argument!

Inside of a Component class definition, all of the passed in props are automatically assigned to the `props` instance variable, accessible as **`this.props`**. The `this.props` variable contains an regular JavaScript *object* whose keys are the props' "names":

```
//Define a component representing information about a user
class UserInfo extends Component {
    render() {
        //access the individual props from inside the `this.props` object
        let userName = this.props.userName;
        let descriptor = this.props.descriptor;

        //can use props or logic or processing
        let userNameUpper = this.props.userName.toUpperCase();

        return (
            <div>
                <h1>{userNameUpper}</h1>
                <p>Hello, my name is {name} and I am a {descriptor}</p>
            </div>
        )
    }
}

let userInfo = <UserInfo userName="Ethel" descriptor="Aardvark" />;
```

Again, *all* props are stored in the `this.props` object—you have to access them through that instance variable.

Note that the `this.props` object is **read only**—meaning that you cannot assign new values to it, nor modify its props directly:

```
//Defines a component representing a message. Contains an error!
class Message extends Component {
    render() {
        this.props.message = "I've changed your message!"; //TypeError!
        return <p>{this.props.message}</p>
    }
}
```

You should just think of props as immutable data that is coming in from outside of the component (again, similar to function arguments). A component doesn't create or change is own props, it just uses those to determine how to render its content.

Importantly, props can be **any** kind of value! This includes basic types such as Numbers or Strings, data structures such as Arrays or Objects, or even functions or other Components!

```
//Props can be of any data type!

//Pass an array as a prop!
let array = [1,2,3,4,5];
let suitcase = <Suitcase luggageCombo={array} />;

//Pass a function as a prop (like a callback)!
function sayHello() {
  console.log('Hello world!');
}
let greeting = <Greeting callback={sayHello} />;

//Pass another Component as a prop (not common)!
let card = <HolidayCard message="Greetings world!" />
let gift = <Gift toMsg="Ethel", fromMsg={card} />
```

Indeed, passing *callback functions* as props to other components is a major step in designing interactive React applications. See the next chapter for more details and examples.

16.4.1 Props and Composition

Recall that React components are usually *composed*, with one Component's `render()` function instantiating other components (as in the `MessageList` example above). Since props are specified when a Component is instantiated, this means that a "parent" component will need to specify the props for its children ("passing the props"). Commonly, the props for the child components will be derived from the props of the parent component:

```
//Defines a component representing a message in a list
```

```
class MessageItem extends Component {
  render() {
    return <li>{this.props.message}</li>
  }
}

//A component that renders a trio of messages
class MessageListTrio extends Component {
  render() {
     return (
       <div>
         <h1>Messages for you</h1>
         <ul>
            {/* instantiate child components, passing data from own props */}
            <MessageItem message={messages[0]} />
            <MessageItem message={messages[1]} />
            <MessageItem message={messages[2]} />
         </ul>
       </div>
     );
  }
}

//Define and pass a prop to the parent comment when rendering it
let messagesArray = ["Hello world", "No borders", "Go huskies!"];
ReactDOM.render(<MessageListTrio messages={messagesArray} />,
                document.getElementById('root'));
```

This creates a **one-directional data flow**—the data values are passed into the parent, which then passes data down to the children (not vice versa).

The Component organization in the previous example is particularly common in React. In order to effectively render a list (array) of data, you should define a Component that declares how to render a *single* element of that list (e.g., `MessageItem`). This allows you to focus on only a single problem, developing a re-usable solution. Then you define another component that represents how to render a whole list of elements (e.g., `MessageList`). This component will render a child item component for each element—it just needs to set up the list, without worrying about what each item looks like!

The data set can be passed into the parent (List) component as a prop, and each individual element from that data set will be passed into an instance of the child (Item) component. In effect, the List component is tasked with with taking the data set (an array of data values) and producing a set Item components (an array of Components). This kind of transformation is a *mapping* operation (each data value is mapped to a Component), and so can easily be done with the **`map()`** method:

```
//Map an array of message strings to an array of components
let msgItems = this.props.messages.map((msgString) => {
    let component = <MessageItem message={msgString} />; //pass prop down!
    return component; //add this new component to resulting array
})
```

And because including an *array of Components* as an inline expression in JSX will render those elements as siblings, it's easy to render this list from within the parent:

```
class MessageList extends Component {
  render() {
     let msgItems = this.props.messages.map((msgString) => {
        let component = <MessageItem message={msgString} />; //pass prop down!
        return component; //add this new component to resulting array
     })

     return (
       <div>
         <h1>Messages for you</h1>
         <ul>
            {/* render the array of MessageList components */}
            {msgItems}
         </ul>
       </div>
     );
  }
}

//Define and pass a prop to the parent comment when rendering it
let messagesArray = ["Hello world", "No borders", "Go huskies!"];
ReactDOM.render(<MessageList messages={messagesArray} />,
                document.getElementById('root'));
```

Note that the above example will issue a warning. This is because React requires you to specify an additional prop called **`key`** for each element in an array of components. The `key` should be a *unique string* for each element in the list, and acts as an identifier for that element (think: what you might give it as an `id` attribute). React will use the `key` to keep track of those elements, so if they change over time (i.e., elements are added or removed from the array) React will be able to more efficiently modify the DOM.

If unspecified, the `key` will default to the element's index in the array, but this is considered unstable (since it will change as items are added or removed). Thus a better approach is to determine a unique value based on the data—which means that data items will need to be uniquely identifiable. The below improved version of the code changes each message from a basic String to an Object that has `content` and `id` properties, allowing each data value to be uniquely identified. Note that the `MessageItem` component need not be changed; it still renders an `<li>` showing the given `message` prop!

```
let messageObjArray = [
    {content: "Hello world", id=1}
    {content: "No borders", id=2}
    {content: "Go huskies!", id=3}
];

class MessageList extends Component {
  render() {
     let msgItems = this.props.messages.map((msgObject) => {
         //return a new MessageItem for each message object
         return <MessageItem
                    message={msgObject.content}
                    key={msgObject.id.toString()}
                    />;
     })

     return (
       <div>
         <h1>Messages for you</h1>
         <ul>
            {msgItems}
         </ul>
       </div>
     );
  }
}

//Define and pass a prop to the parent comment when rendering it
ReactDOM.render(<MessageList messages={messageObjArray} />,
                document.getElementById('root'));
```

This kind of pattern enables you to effectively define your web page's content in terms of individual components that can be composed together, allowing you to quickly render large amounts of data (without needing spend a lot of time thinking about loops!).

Resources

As mentioned above, there are a large number of different resources for learning React. Below are links to key parts of the official documentation.

- Thinking in React

- Facebook's React Tutorial
- React Documentation
- JSX Documentation
- Components and Props
- React API References
- Create React App Docs

Chapter 17

React Tools

React and other modern JavaScript development approaches require leveraging a variety of external libraries, and often require other compilation steps such as **transpiling** to ES6, or **bundling** your JavaScript into a single file. There are a variety of tools available to make these steps easier, some of which we'll introduce here.

17.1 Importing and Exporting in ES6

One of the most powerful features of ES6 is the ability to import and export functions from files. This allows you to create reusable blocks of code that are easily integrated to multiple projects. You no longer need to consider if a different JavaScript file has been loaded into the `index.html` file. Instead, we can import desired functions directly into our scripts. Moreover, it means that you don't need to include an entire library – you can just import the functions you want, and leave out the rest. This will help prevent namespace collisions and limit the file-size of your project.

This article provides us with a working definition of *JavaScript Modules*:

> JavaScript modules allow us to chunk our code into separate files inside our project or to use open source modules that we can install via npm. Writing your code in modules helps with organization, maintenance, testing, and most importantly, dependency management.

We'll begin by writing a simple function that we can *export* – this will make the functions available to other scripts that *import* them:

```
// In our utility.js file, write a function that converts feet to meters
function feetToMeters(feet) {
    return feet / 3.28084
}

// Write another function metersToFeet
function metersToFeet(meters) {
    return meters * 3.28084
}

// Export each named function
export {feetToMeters, metersToFeet} // named exports
```

If we then wish to use `feetToMeters` in another function, we can simply import it:

```
// In our main.js file, import the feetToMeters function
import {feetToMeters} from './utility' // assuming we're in the same directory
```

17.1.1 Named v.s. default exports

The above example demonstrates the use of *named* exports, in which we explicitly name each object we wish to export, and then explicitly name it to import it.

```
// Utility.js ----------
// Write a function that converts feet to meters
// In our utility.js file, write a function that converts feet to meters
function feetToMeters(feet) {
    return feet / 3.28084
}

// Export each named function
export {feetToMeters} // named exports
```

There are then three syntax options for *importing* **named exports**:

```
// Importing options for a named export (see above)

// Option 1 -------- import named exports by name
import {feetToMeters, metersToFeet} from './Utility';
let meters = feetToMeters(30); // use function


// Option 2 -------- import named exports by name AS a shorter name
import feetToMeters as f2m from './Utility'
let meters = f2m(30); // use function
```

```
// Option 3 -------- import all named exports with a prefix
// Import our own components
import * as Utilities from './Utility';
let meters = Utilities.feetToMeters(30); // use function
```

Alternatively, you can export a **single default object** from a module:

```
// Utility.js ----------
// In our utility.js file, write a function that converts feet to meters
function feetToMeters(feet) {
    return feet / 3.28084
}

// Export a default module
export default feetToMeters; // single default export
```

Then, to import the **default export**, you can use the following syntax:

```
import feetToMeters from './Utility.js'
```

You'll use this syntax to import // export your own functions, and you'll also use them to import functions from **external libraries**.

17.2 Installing with NPM

We introduced the use of NPM in an earlier chapter, largely to install packages that were described in the `package.json` file. As we move forward, we'll use NPM to **install additional packages** and **deploy our code**.

It's a common occurrence that, part way through a project, you identify **additional libraries** that you would like to use in your project. You can use NPM to install a package and **add it to your list of packages** if you use the `--save` option:

```
# Download the lodash library, and add it to your list of packages
npm install lodash --save  # make sure to run this in your project folder
```

Adding it to your list of packages enables others to install the necessary files to use your program. When you start using a compiling program (like the functionality built into `create-react-app`) you'll be able to import functionality from your `node_modules` folder as you would other functions (i.e., `import`):

```
// In a JS file in a project that gets compiled (using create-react-app)
import {uniq} from 'lodash';   // import a named export
let vals = uniq([1, 2, 3, 1]); // returns [1, 2, 3]
```

17.3 Create-React-App

In order to leverage a more complex structure for our code (such as importing and exporting functions), we need to **compile our JavaScript** before loading it into the browser. There are a variety of tools that people use to do this such as Gulp, Browserify, and Webpack. These tools are great for configuring a build system for your project, but they can be time-consuming to configure (especially if you don't have any custom specifications that you're trying to make).

So, the folks over at React built an amazing tool called create-react-app, which combines multiple tools (including Webpack) to compile your code for you. It's super easy to get started:

```
# Globally install the create-react-app command line utility: only do this once!
# Depending on your permissions on your machine, you might need to `sudo npm install
npm install -g create-react-app

# Create a project called my-app (a child of your current directory)
create-react-app my-app

# Change your directory to my-app
cd my-app/

# Start running a local server with your project: visible at localhost:3000
npm start  # this functionality is defined in the package.json file that was created
```

Create React App is somewhat opinionated about the structure of your code, but comes with a number of advantages. For example, it will display **errors and warnings in your browser** window, will **automatically reload** when your JavaScript changes, and will **inject CSS changes** without reloading your page. When you create a new project, the following files will be created:

```
my-app/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
```

From there, you can begin editing your `App.js` file. Your `<App/>` component is rendered on the page, so you should augment it as you see fit. You can now import functions directly into a JavaScript file from libraries **or** your own modules:

```
// Import desired files
import {Component} from 'react' // import the Component class from the react library
import {uniq} from 'lodash';    // import a named export from the lodash library

// Create a class App by extending Component
class App extends Component {
  render() {
    return <div>My App</div>
  }
}

// Export the App component to be imported in other files
export default App;
```

17.3.1 Building

In order to build your app for deployment, you will use the `npm run build` command (you can see what function this does by looking at the `package.json` file). This will appropriately compile your files (including JavaScript, CSS, images, etc.) into an optimized format. The only tricky thing is if you're trying to host your build at a particular location, you'll need to **specify that location** in your `package.json` file. For example, imagine you're going to host your site at `http://students.washington.edu/YOUR-NAME/info-343/project-name`. Create React App assumes that you'll be hosting your project in the **root of your server**. In other words, when it builds, it will construct relative paths that assume resources are at the root. To adjust this, you'll need to edit your `package.json` file:

```
{
  "name": "project-name",
  "version": "0.1.0",
  "private": true,
  "homepage": "http://students.washington.edu/YOUR-NAME/info-343/project-name",
  .....
}
```

By setting the `homepage`, the relative paths will be properly constructed. This will create for you a folder called `build/`, in which you'll have the necessary files. You can then **move these up a directory** (i.e., out of the `build/` folder) if you're going to use your GitHub repo for hosting (this is an unfortunate manual step). Then you should be ready to deploy!

Another option is to use the `gh-pages` library. This will provide you with a command line tool to create a `gh-pages` branch, and move the files in your `build/` folder up a directory (meaning that you don't need to go to the URL `project-nanme/build/`). Here are the steps for using the `gh-pages` package:

```
# Install the gh-pages package in your project
npm install gh-pages --save
```

Then, edit your `package.json` file to **add a new command** in the `scripts` section. You'll add the `"deploy"` line of code.

```
"scripts": {
     "start": "react-scripts start",
     "build": "react-scripts build",
     "test": "react-scripts test --env=jsdom",
     "eject": "react-scripts eject",
     "deploy": "npm run build&&gh-pages -d build"
 }
```

Then, to deploy a project to the `gh-pages`*branch* on GitHub, you can simply run this line of code (this will `build` your project, and `commit` and `push` the built project to your branch):

```
npm run deploy
```

Then, on your server, you should pull down the code and checkout your `gh-pages` branch:

```
# On your server, such as a UW student server
git clone PROJECT-URL
git checkout gh-pages

# To update:
git pull origin gh-pages
```

17.4 Resources

- React Create App
- Export Documentation
- Import Documentation
- Getting Started with WebPack
- Node Package Manage (NPM) Documentation
- Node Download Page
- Using a Package.json File
- An Intro to ES6 and NPM

Chapter 18

Interactive React

This chapter describes how React components can be used to build interactive and robust applications. It describes how you can store information in a Component's **state**, assign **event** handlers, and leverage the **component lifecycle** to trigger re-rendering of your application.

18.1 State

So far, we've been discussing how components can receive *properties* to determine their visual rendering. It's put nicely in this write-up:

> *props* (short for properties) are a Component's configuration, its options if you may. They are received from above and immutable as far as the Component receiving them is concerned. A Component **cannot change its props**, but it is responsible for putting together the props of its child Components.

So, we use **props** to pass data into components which render that data. However, if we want to begin building dynamic applications, we'll need to use **component state** to track change in the *state* of the application (usually driven by a user).

A really nice overview of the difference between props and state is in this article, which points out a few key ideas:

- "State is reserved only for interactivity, that is, data that changes over time"
- Keep your state *as minimal as possible*, and compute information from that state

So, for example, if you wanted to have a *searchable list*, that component would recieve the list items as an array (which would not change) – this means that it is part of **props**. However, the search string will change, so we can store the search string in **state**, and *compute the filtered list* in our `render()` function.

Note: one of the most difficult parts of architecting a React application is figuring out what information should be stored as props or state (and in which Components). We suggest you practice this using simple examples, and think carefully about it as you build more robust applications.

We can begin by setting an initial state in the class's `constructor`

```
// Component for searching a table
class SearchableTable extends React.Component {
    // Set an initial state: it will then be accessible as this.state.NAME
    constructor(props) {
        super(props); // exposes props to this.props in the constructor
        this.state = {
            search:''
        };
    }
    render() {
        // Use searchstring to filter data
        ...

        // Render a table in here
        return(...dom elements ...);
        ...
    }
});
```

If we were building a searchable table, the only thing that changes, and therefore the **only data we need to store in state**, is the search string – we can then filter down our data in the `render` function.

18.2 Component Lifecycle

The true benefit of using React components to build your application is that **lifecycle events** trigger a *re-rendering* of the component. As a result, you no longer need to concern yourself with assigning event handlers to update your DOM. Instead, you simply need to **change the data** (*state* or *props*) of your component, and React will handle the re-rendering.

As described in the documentation, there are a set of events that are triggered when a component is **mounted** on the DOM, when the component's data is **updated**, and when the component **unmounts**. Each one of the lifecycle events will be triggered automatically at the appropriate time, and you can define the desired behavior as a method on your class. For example, it's quite common to **load data** when the component is mounted. Here's an example using the `d3-request` package to load a `.csv` file.

```
// Import the csv loading function and Component
import {Component} from 'react';
import { csv } from 'd3-request';

// Define class App by extending Component
class App extends Component {
    constructor(props) {
        super(props);

        // Set initial data as an empty array
        this.state = {
            data:[]
        }
    }

    // Load data when component mounts: this will trigger automatically
    componentDidMount() {
        // Use the csv method to load the data, then set the state
        csv('data/all_data.csv', (error, data) => {

            // Set the stated: this will trigger a re-rendering of the app
            this.setState({
                data: data
            });
        })
    }

    // render method
    render() {
        // Do something with your data in here....
        return <div>My App</div>
    }
}
```

In the above code, the `componentDidMount` method will trigger when the component is mounted to the dom. In the method, the state will be updated (`this.setState()`), which will in turn re-render the `App` component. You can write custom event handlers for *any* of the lifecycle methods, enabling you to have granular control over your application.

18.3 Events

Event handlers are assigned in React in the same way that you would assign them to an HTML element using the event name of your choice. Inside of a `.jsx` file, we could leverage the curly braces `{}` inside an HTML section to reference a JavaScript function.

```
<input onChange={functionName} />
```

What we need to figure out is *what to do in that function*. Rather than trying to update elements ourselves, we can simply **update the state**, and let React take care of the rest. When we change state or props, React will re-render our components. Continuing with our example of a searchable table, we could define a function *as part of our component* that *changes it's state*, and then our UI will be re-rendered:

```
// Component for searching a table
class SearchApp extends React.Component {
    // Set an initial state: it will then be accessible as this.state.NAME
    constructor(porps) {
        super(props);
        this.state = {
            search:''
        };
    },
    // Define a `filter` function to be executed when the input element changes
    update(event) {
        // Get event value
        let value = event.target.value;

        // Change state
        this.setState({search:value});
    }
    render() {
        // Use searchstring to filter data
        ...

        // Render a table in here
        return(
            <input onChange={this.update} />
            ...other dom elements...
        );
        ...
    }
});
```

In the above section, the `filter` function (which we define – this **is not** a default React function) will be executed when the `<input>` element changes value. The `filter` function then set's the state using `this.setState({key:value})`. Note **do not** try setting the state directly (i.e., `this.state.searchString = 'something'`). By setting the state, you will **trigger an update**, and React will re-render your DOM. For more information on events that trigger a re-render, see this State and Lifecycle article. For more information on events, see here.

18.4 Lifting Up State

In architecting a React application, you'll need to make the following design choices:

- What information is tracked as *props* v.s. *state*?
- Which components track the *state* v.s simply receiving *props*?
- How can you pass information across components?

A simple model for React applications is to have a singe `<App>` Component that tracks the state. That application will pass all information to the other components via *props*. However, this raises a difficult question:

> Given React's *one-directional* data flow, how can you pass information *back to the* `<App>` from a child Component?

For example, if we are building a `SearchApp` that is a searchable table, it may have an `<UserInput>` Component where the user is typing:

```
class SearchApp extends React.Component {
    render() {
        return (
            <div>
                <UserInput/>
                <Table/>
            </div>
        )
    }
}
```

The process of passing information *back up to* the `SearchApp` from the `UserInput` component is know as Lifting State Up. To accomplish this, we'll need to pass an **event handler** from our parent (`SearchApp`) to the child (`UserInput`). This will allow the parent to register changes when the *child* experiences events. For example, a `UserInput` element could be structured to *recieve a function as props* that it will execute `onChange`:

```
class UserInput extends React.Component {
    render() {
        return (
            <div>
                <input onChange={this.props.update}/>
            </div>
        )
    }
}
```

The `UserInput` is build to be flexible: it will do whatever function is passed in as `update`. So, we can define a function in our `SearchApp` and pass it to the `UserInput` as a property:

```
class SearchApp extends React.Component {
    handleChange(event) {
        // Get event value
        let searchValue = event.target.value;

        // Set the state to trigger a re-rendering
        this.setState({search:searchValue})
    }
    render() {
        // Set the `update` property of the `UserInput` element
        return (
            <div>
                <UserInput update={this.handleChange}/>
                <Table/>
            </div>
        )
    }
}
```

To see a working example of this application, see this codepen.

Chapter 19

Client-Side Routing

This chapter discusses how to use React to effectively develop **Single Page Applications (SPA)**—web applications that are located on a single web page (HTML file), but use AJAX requests and DOM manipulation to produce the *appearance* of multiple "web pages". This structure is facilitated by the use of the *client-side routing* library `react-router`, which allows you to render different Components based on the browser's URL, allowing each View ("page") to be treated as a unique resource.

19.1 Single-Page Applications

As you've seen in previous chapters, the React framework lets you dynamically render different *Views* (Components) based on different conditions such as the `state` of the app. For example, you can have a blogging app that could have a `this.state.blogPostId` variable, and then use that variable to determine which blog post to display. Often these Views act as entirely separate **pages**—you either show one View or an another. As such, you'd often like each View to be treated as an individual *resource* and so to have its own **URI**, thus allowing each View to be referenced individually. For example, each blog post could have it's own URI, allowing a user to type in a particular URL to see a specific post (and letting that user share the post with others).

In order to achieve this effect, you can utilize **client-side routing**. With client-side routing, determining which View to display based on the URL (how to "route", or map that URL to the correct resource) is performed on the *client-side* by JavaScript code. This is distinct from *server-side routing*, in that the server isn't deciding which resource to show (i.e., which `.html` file to respond to a request with), but rather responds with a single HTML file whose JavaScript dynamically determines what resource to show (i.e., which React component to render) based on the *URI* that request was sent to!

- In this context, "routing" involves taking the resource identifier (the URI) and determining what *representation* of that resource should be displayed—what View to show. A "route" is thus a URI, which will refer to a particular View of the resource.

Client-side routing allows you to have unique URLs for each View, but will also make the app work faster—instead of needing to download an entire brand new page from the server, you only need to download the requisite extra data (using an AJAX request), with much of the other content (the HTML, CSS, etc) already being in place. Moreover, this will all your app to easily share both state data and particular components (e.g., headers, navigation, etc).

- Google Drive is a good example of a Single-Page Application. Notice how if you navigate to a new folder, the URL changes (so you can link to individual folders), but only a single "pane" of the page changes.

Because React applications are component-based, you can perform *client-side routing* in React by using conditional rendering to only render components if the current route is correct. This follows a structure similar to:

```
class App extends Component {
  render() {
    //pick a component based on the URL
    let componentToRender = null;
    if(currentUrl === 'domain/home'){ //pseudocode comparison with URL
      componentToRender = <HomePage />;
    }
    else if(currentUrl === 'domain/about'){
      componentToRender = <AboutPage />;
    }
    //render that component
    return componentToRender;
  }
}
```

- That is, *if* the current URL **matches** a particular route, then the Component will be rendered.

19.2 React-Router

Third-party libraries such as **React Router** provide Components that include this functionality, allowing you to easily develop single-page applications.

This chapter details how to use **version 4** of React Router, released in *March*

2017. This version is significantly different from the previous versions (2.x and 3.x). Be careful when looking up examples and resources that they're utilizing the same version as you!

As with other libraries, you begin using React Router by installing the `react-router-dom` library (the browser-specific version of React Router):

```
npm install --save react-router-dom
```

You will then need to `import` any Components you wish to use into the `.js` files containing your React code. For example:

```
//import BrowserRouter (but call it `Router`), Route, and Link
import { BrowserRouter as Router, Route, Link} from 'react-router-dom'
```

These Components are described in the following sections.

Routing

The **`<BrowserRouter>`** Component (which is often imported with an alias, causing it to be instantiated as `<Router>`) is the "base" Component used by React Router. This Component does all the work of keeping the React app's UI (e.g., which Components are rendered) in sync with the browser's URL. The `BrowserRouter` "listens" for changes to the URL, and then slightly passes information about the current route (called the **`path`**) to its child components as a `prop`. This allows each child to always know what route is currently shown in the URL, without needing to access it directly.

- With React Router, a "route" is defined by the *path* portion of a URI (see Chapter 2). This is the part that comes *after* the protocol and domain (e.g., after the `https://mydomain.com/`). Thus the `/home` route would refer to the URI `https://mydomain.com/home`, while the `/about` route would refer to the URI `https://mydomain.com/about`.

- `BrowserRouter` utilizes the HTML5 history API to interact with the brower's URL and history (what allows you to go "back" and "forward" between URLs). This API is supported by modern browsers, but older browsers (i.e., IE 9) would need to use `<HashRouter>` as an drop-in alternate. `HashRouter` uses the fragment identifier portion of the URI to track what "page" the app should be showing, causing URL's to include an extra hash `#` symbol in them (e.g., `https://mydomain.com/#/about`).

Inside (as a child of) the `<BrowserRouter>`, you can specify route-based views using the **`<Route>`** Component. This Component will render some content (a `component`) only when the URL *matches* a specified **`path`**. In effect, the `Route` Component handles checking `if` the current URL matches the specified path, and if so renders the specified Component (similar to in the pseudo-code example above). If the URL doesn't match the route, then the Component is not rendered. Both the `path` to match and the `component` to render are passed in as props:

```
class App extends Component {
  render() {
    return (
      <BrowserRouter>
        {/* if currentUrl == '/home', render <HomePage> */}
        <Route path='/home' component={HomePage} />

        {/* if currentUrl == '/about', render <AboutPage> */}
        <Route path='/about' component={AboutPage} />
      </BrowserRouter>
    );
  }
}
```

The **`path`** prop is used to indicate the route that you wish to match. This route should *always* start with a leading `/` (since it's the path that comes after the domain in the URI). Note that this can be a multi-part path (e.g., `/assignments/react`), and can even include URL parameters (see below).

- Note that by default the `path` will "match" even if it is contained in only part of the URL. For example, `<Route path='/about' />` will match a URL of `/about` *OR* `/about/me`. A path of `/` will match *any* URL! You can customize what how strict the matching is by specifying the `exact` prop (indicating that the path has to match entirely) and/or the `strict` prop, which will cause the router to respect any trailing `/` you include.

- Importantly, each `<Route>` determines whether it should render its component *independently* from each other: they are each `if` statements, not `if else` statements! Thus it is possible for more than one `<Route>` to match the current URL and render its component (and you may actually want to do this sometimes, if a Component is only a part of a "page"). However, it is very common to have each "route" be mutually exclusive so that only one "page" is shown at a time. You can enforce this by nesting the `<Route>` elements inside of a `<Switch>` element:

  ```
  <BrowserRouter>
    <Switch>
      <Route path='/home' component={HomePage} />
      <Route path='/about' component={AboutPage} />
    </Switch>
  </BrowserRouter>
  ```

  This can be particularly useful when working with URL Parameters.

  Pro tip: It is often useful to specify the routes as a `const` variable (e.g., `routes`) that is an object containing paths and which component to render for that path. Then you can use a `map()` operation to render those `<Route>` elements. This makes it easy to check and change the URIs used in your page later. See Route Config for an example.

The **`component`** prop is used to specify *which* Component should be rendered if the route matches. The component is specified by name as an inline JSX expression (so inside `{}`); you're actually passing a *reference to the class* to the `<Router>`, so it can then instantiate that class!

- The rendered Component (e.g., `HomePage` or `AboutPage`) will be passed a few different props from the `<Router>` which give it some context about the current route match that caused it to be rendered. However, you may notice that there is no where in this syntax to specify a prop that *you* may want to pass to the element (e.g., something from the `state` such as the current logged in user).

  In order to pass in your own props, you specify a `render` prop instead of the `component` prop. This prop takes in a *callback function* which will be passed in the "router props", to which you can then add in your own props:

  ```
  <Route path='/home' render={(routerProps) => (
      {/* use spread operator to convert the object into individual props */]}
      <HomePage {...routerProps} myMessageProp={"Hello World"} />
  )}
  ```

  In general, you should utilize the `component` prop instead of `render`. It is cleaner, and helps to keep the page Components self-contained and "separated".

URL Parameters

It is also possible to include *variables* in the matched route using what are called **URL Parameters**. As you may recall from reading a RESTful API, URI endpoints are often specified with "variables" written using **`:param`** syntax (a colon `:` followed by the parameter name). For example, the URI

```
https://api.github.com/users/:username
```

from the Github API refers to a particular user—you can *replace* `:username` with any value you want: `https://api.github.com/users/joelwross` refers to the `joelwross` user, while `https://api.github.com/users/mkfreeman` refers to the `mkfreeman` user.

React Router supports a similar syntax when specifying Route paths. For example:

```
<Route path='/post/:postId' component={BlogPost} />
```

will match a path that starts with `/post/` and is followed by any other path segment (e.g., `/post/hello`, `/post/2017-10-31`, etc). The `:postId` (because it starts with the leading `:`) will be treated as a parameter which will be assigned whatever value is part of the URI in that spot—so `/post/hello` would have `'hello'` as the `postId`, and `/post/2017-10-31` would have `'2017-10-31'` as the `postId`.

The value assigned to the URL parameter will be passed to the rendered component (e.g., `<BlogPost>` in the above example) as a part of the **`match` prop**, which is one of the "router props" that the `<Router>` element passes into its `component`.

As such, the value of the URL parameters will be available to the rendered Component as **`this.props.match.params.paramName`**. The rendered component can then use this prop to determine what content to render, perhaps accessing that data from a Model module.

```
class BlogPost extends Component {
  render() {
    return (
        <h1>You are looking at blog post {this.props.match.params.postId}</h1>
    )
  }
}
```

Nesting Routes

As you're working with React Router, remember that `<Route>` elements are *just React Components* (that include an `if` statement causing them not to render if the URL is incorrect). That means that—as long as they are inside a `<Router>`, you can include them anywhere inside your application, mixing them in with normal HTML and React Components:

```
class App extends Component {
  render() {
    return (
      <div>
        <header>
          <h1>Page Title</h1>
        </header>
        <BrowserRouter>
          <Route path='/home' component={HomePage} />
          <Route path='/about' component={AboutPage} />
        </BrowserRouter>
      </div>
    );
  }
}

class HomePage extends Component {
  render() {
    return (
      <div>
        <h2>Home Page</h2>
        {/* if route includes /blog, show the blog */}
        <Route path={this.props.match.url + '/blog'} component={BlogPartial} />
      </div>
    );
  }
}
```

Linking

While specifying `<Route>` elements will allow you to show different "pages" at different URLs, in order for a Single Page Application to function you need to be able to *navigate between routes* without causing the page to reload. Thus you can't just use normal `<a>` elements to link between "pages"—browsers interpret clicking on `<a>` elements as a command to send a new HTTP request, and you instead just want to change the URL and re-render the App.

Instead, React Router provides a **`<Link>`** element that you can use to create a hyperlink to another route within the application. This component takes **`to`** prop that you use to specify the route that it links to:

```
<Link to='/about'>Click to visit the About Page</Link>
```

- The component will render as an `<a>` element with a special `onClick` handler that keeps the browser from loading a new page. Thus you can specify an content that you would put in the `<a>` (such as the hyperlink text) as child content of the `<Link>`.
- It is also possible to specify additional parts (e.g., query parameters, fragments) as part of the link. See the documentation for details.
- React Router also provides a `<NavLink>` Component that lets you specify a specific CSS class or styling that should apply to the element if the `to` route matches the *current route*. This is used for example to have a navigation section "highlight" the link to the page you're currently on, helping the user understand where they are on the page.

Finally, React Router provides a **`<Redirect>`** component that, when rendered, will navigate the browser to the given route. This will allow you to "programmatically" navigate the user to a different route (without requiring the user to click on a link)—you just need to render the `<Redirect>` and the page will change.

- Note that in order for the `<Redirect>` to work, you need to *render* it (e.g., return it from a Component's `render()` function). Thus a good way to programmatically redirect is to specify a `state` variable (e.g., `shouldRedirect`), and then conditionally render the `<Redirect>` if that state variable becomes true:

```
class LoginPage extends Component {
  constructor(props) {
    super(props)
    this.state = {};
  }

  componentDidMount() {
    //hypothetical method to check if user is logged in
    checkAuthenticationState().then((status) => {
      if(status === LOGGED_IN) //user is logged in
        this.setState({redirect: true}); //re-render, but redirect
    })
  }

  render() {
    if(this.state.redirect) { //if we should redirect
      return <Redirect to='/home' />;
    }

    //otherwise, show the form
    return (
      <div>
        <form class="login-form">
          ...
        </form>
      </div>
    );
  }
}
```

Caution: you should *not* render a `<Redirect>` element as the child of displayed content (e.g., inside a `<div>`). This can cause issues with the redirect taking multiple "DOM update cycles" to process, interfering with your application's processing. Instead, determine whether you should redirect and if so `return` just the `<Redirect>` element (e.g., a "break early" sentinel condition).

That covers most of the basic features of React Router. Be sure to check out the documentation for more details, as well as the extensive examples (though they use some alternate, less readable React syntax).

React Router and Github Pages

React Router's client-side routing introduce a few additional considerations when the you wish to deploy your app on a non-development server, such as Github Pages (e.g., what happens when you deploy a `create-react-app` project).

First, consider what happens when you type a route (e.g., `https://domain.com/about` to access the `/about` route) into the browser's URL bar in order to navigate to it. This creates an HTTP Request for the resource at the URI with an `/about` path. When that request is received by the web server, that server will perform *server-side routing* and attempt to access the resource at that location (e.g., it will look for an `/about/index.html` page). But this isn't what you want to happen&dmdash;because there is no content at that resource (no `/about/index.html`), the server will return a 404 error.

Instead, you want the server to take the request for the `/about` resource and *instead* return your root `/index.html` page, but with the appropriate JavaScript code which will allow the *client-side routing* to change the browser's URL bar and show the content at the `/about` route. In effect, you want the server to be able to return your root `index.html` page no matter what route is specified in the HTTP Request!

It is perfectly possible to have a web server do this (to *not* perform server-side routing and instead always return `/index.html` no matter what resource is requested); indeed, this is what the Create React App development server does. However GitHub Pages doesn't have this functionality: if you send an HTTP request for a resource that doesn't exist (e.g., `/about`), you will receive a 404 error. There are a few ways to work around this:

1. You can utilize a **`<HashRouter>`** instead of a `<BrowserRouter>` The `<HashRouter>` uses the fragment identifier portion of the URI to record and track which route the user is viewing: the HTTP request is thus sent to `https://domain.com/index.html#/about` to get the `/about` route—and since `index.html` is the default resource, this can be abbreviated to `https://domain.com/#/about`, which is *almost* as good. In this way you are always requesting the appropriate resource (`/index.html`), but can still perform client-side routing. The trade-off is that your URLs will have extraneous `#` symbols in them (which also makes utilizing inner-page navigation with the fragment more difficult), and going to `domain.com/about` will *still* cause a 404 error.

2. The other approach is to replace Github Page's 404 page with something that goes to your `index.html` (using *server-side routing*)—so instead of the user being shown the 404, they are shown your `index.html` which is about to do the client-side routing! `spa-github-pages` provides some boilerplate for doing this, but it is a "hacky" approach as is not recommended.

3. The best approach would be to utilize a different web hosting system that better supports the server-side routing needed for single-page applications. For example, Firebase Hosting allows you to specify a rewrite rule that will cause the server to return your `index.html` no matter which route the HTTP Request specifies. Create React App also has some details about deploying to Firebase.

Second, in addition to the server-side routing issue, you will need to do extra work to handle `<Redirect>` elements if your application's URL is in a *subresource* of the server (e.g., it can be found at `https://domain.com/app/index.html`). While `<Link>` elements will route correctly in this case (`<Link to='/home'>` will go to `https://domain.com/app/home`), `<Redirect>` elements will *overwrite* the path part of the URI: `<Redirect to='/home'>` will take you to to `https://domain.com/home`, losing the fact that your application is in `/app` resource.

Luckily, it is easy to support this behavior and tell React Router that all paths should be treated relative to that subresource (relative to the `/app` path). You do this by passing the **`basename`** prop to the `<Router>`:

```
<BrowserRouter basename={process.env.PUBLIC_URL+'/'}>
```

- This example specifies that the "base" uri should be whatever URI was listed in the `"homepage"` key of the project's `package.json` folder, such as what you specifying when deploying Create React App to Github Pages. You would want to use that exact expression (`process.env.PUBLIC_URL`), which is a global variable referring to the _env_ironment of the bundling node process; the `PUBLIC_URL` key is assigned the `homepage` property by Create React App.

Resources

- React Router Official Guide - starts with "Philosophy" that explains how it is used
- React Router API - complete list of Components and props
- Accessible React Navigation - a blog post on making routing accessible

Chapter 20

Firebase

This chapter discusses how to integrate and utilize the **Firebase** web service into a client-side application (using React). Firebase is a web service that provides tools and infrastructure for use in creating web and mobile apps that store data online in the cloud (a "web backend solution"). In effect, Firebase acts as a *server* that can host information for us, so that you can just build client-side applications without needing to program a web server.

- Firebase is owned and maintained by Google, but will still work perfectly fine with Facebook's React framework.

In particular, this chapter will discuss two features of Firebase:

1. The Firebase service offers a client-side based system for performing **user authentication**, or allowing users to "sign in" to your application and have particular information associated with them. It's possible to have users sign up with an email and password, or even using an external service (e.g., "sign up with Facebook"). Firebase also provides a way to manage all those extra interactions associated with accounts, like being able to reset passwords or confirm email addresses. And this is all performed securely from the client-side, without the need to set up an additional server to perform OAuth work.

2. The Firebase service also provide a **realtime database**, which provide a cloud-hosted database for persisting data between page visits. Firebase's database is a NoSql-style database: you can think of it as a *single giant JSON object in the Cloud*. Firebase provides methods that can be used to refer to this object and its properties, make changes to the object, and even "listen" for changes made by others so that the page can automatically update based on actions taken by other users. In effect you can create a **data binding** between clients and the server, so that if you update data on one machine, that change automatically appears on another.

These features mean that Firebase can be a go-to back-end for producing a client-side app that involves persisted data (like user accounts). And its free service tier is perfectly suited for any development app.

Important note: Firebase updated to version 3.0 in **May 2016**. This was a massive revision to the API and how you should interact with the service. Watch out for out-dated "legacy" examples. when looking for help.

20.1 Setting up Firebase

Because Firebase is a cloud service, you will need to set it up externally in order to use it in your web application. Firebase setup and configuration is handled on Firebase's website at https://firebase.google.com/, and in particular in the Firebase Web Console where you can manage individual projects.

When developing a Firebase app, you will want to keep the Web Console open so you can check on the user list and database.

Creating a Project

In order to use Firebase in your web app, you will need to *sign up for the service*. Visit **https://firebase.google.com/** and click the "Get Started" button to do so. You will need to sign in with a Google account (e.g., a UW account if you've set up Google integration). Signing up will direct to the Firebase Web Console.

In the Web Console, you can manage all of your "projects"—one per application you've created. Each project will track its own set of users and have its own database of information.

You can create a project by clicking the *"Add Project"* button. In the pop-up window that appears, you'll need to give the app a *unique* name. Naming the project after your app is a good idea. This name is used internally by the Firebase system (it won't necessarily be visible to your users).

Once you've created the project, you will be taken to the **Web Console** for that project. This is a web page where you will be able to manage the configuration of the project, the users who have signed up, and any data in the database. In particular, note the navigation menu on the left-hand side: you will use this to access different parts of your project (*Authentication* to manage users, and *Database* to manage the database).

Including Firebase in React

In order to use Firebase in a web app (including a React app), you will need to add the Firebase library to your web page, as well as some specify some

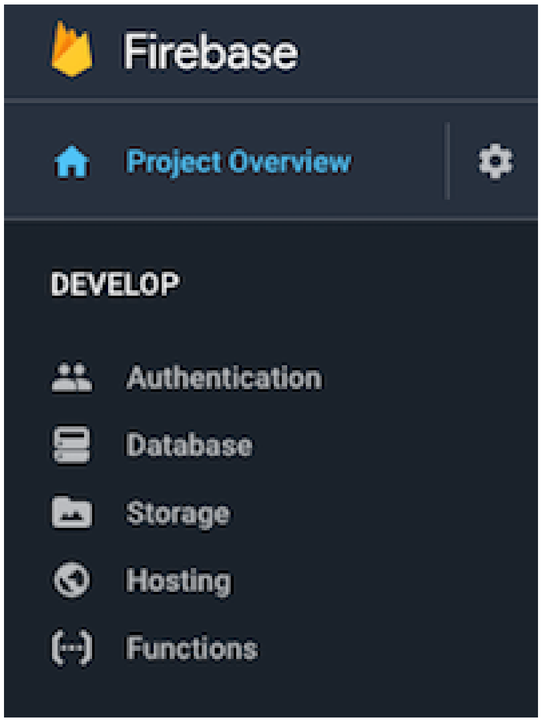


Figure 20.1: Firebase navigation menu.

configuration data to connect to the correct Firebase project.

From the project's Web Console page, click the *"Add Firebase to your web app"* button. This will produce a popup with some HTML code for loading the Firebase library, as well as an inline `<script>` that configures your app to use Firebase.

However, in a React app you should integrate Firebase directly into your source code instead of modifying the HTML. You can do this by modifying the **`index.js`** file:

1. First, install and Firebase library using `npm`:

   ```
   # in your app repo
   npm install firebase
   ```

2. `import` the Firebase library into your `index.js` file. To only load the core firebase functionality, you can just import the `'firebase/app'` module.

   ```
   import firebase from 'firebase/app';
   ```

   You can also `import` modules with the specific pieces of functionality you need, such as user authentication and the real-time database.

   ```
   import 'firebase/auth';
   import 'firebase/database';
   ```

   (These modules don't export any need functions beyond modifications to the core Firebase app, so you don't need to assign a variable name to the resulting value).

3. To configure your web app, **copy and paste** the contents of the `<script>` file shown on the Firebase Web Console (The `// Initialize Firebase ...` part). Do this *before* the `ReactDOM.render()` call.

   This will specify *which* Firebase project your web app should connect to.

20.2 User Authentication

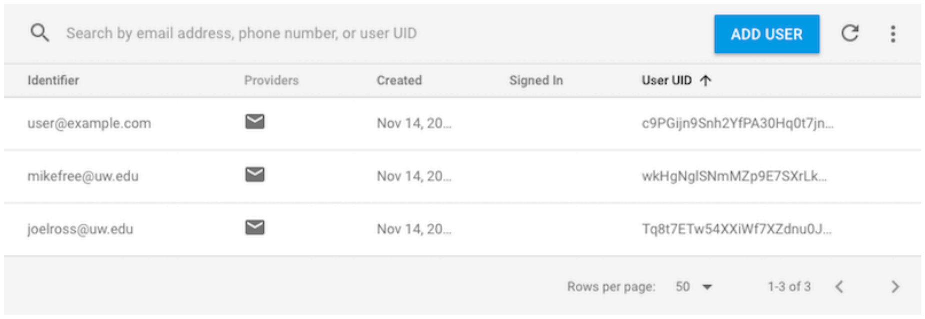
Firebase provides the ability to **authenticate** users: to let people sign up for you application and then to check that they are who they say (e.g., have provided the right password). Firebase supports multiple different forms of authentication: users can sign up with an email and password, log in with a social media service such as Google or Facebook, or even authenticate "anonymous" (so you can at least keep track of different people even if you don't know who they are).

In order to support user authentication, you will need to enable this feature in the Firebase Web Console. Click on the "Authentication" link in the side navigation menu to go to the authentication management page. Under the

"Sign-In Method" tab, you can choose what forms of authentication you wish to enable.

- For example, click on the "Email/Password" option, and then flip the switch to "Enable" that method. Be sure and Save your changes!

Note that you will be able to use this page to see and manage users who have signed up with your Firebase project. This is where you can see if uses have successfully been created, look up their **User UID** for debugging, or delete users from the project:



Identifier	Providers	Created	Signed In	User UID ↑
user@example.com	✉	Nov 14, 20...		c9PGijn9Snh2YfPA30Hq0t7jn...
mikefree@uw.edu	✉	Nov 14, 20...		wkHgNglSNmMZp9E7SXrLk...
joelross@uw.edu	✉	Nov 14, 20...		Tq8t7ETw54XXiWf7XZdnu0J...

Rows per page: 50 1-3 of 3

Figure 20.2: Firebase users.

Creating Users

In order to create a user programmatically (e.g., from JavaScript when the user submits a sign-up form), you can use a function provided by the `firebase` module. Specifically, you'll call the wordily-named function `firebase.auth().createUserWithEmailAndPassword` passing it the email and password the user is signing up with. This will create a new user account in Firebase (you can view it in the Firebase Web Console), *as well as* log in the user.

```
firebase.auth().createUserWithEmailAndPassword(email, password)
    .then((userCredentials) => {
        let user = userCredentials.user; //access the newly created user
        console.log('User created: '+user.uid);
        //...
    })
    .catch((error) => { //report any errors
        console.log(error.message);
    });
```

- Remember to `import firebase from 'firebase/app'` in any module that you need to access the `firebase` global variable!

- The `createUser...` method is called on `firebase.auth()`. The `auth()` function returns an "authenticator" object that can manage user login which gets an "authenticator" object that is connected to the web service and can manage user login information.
- The `createUser...` method returns a *Promise*, so you can use the `.then()` method to do further work after the user is created (e.g., logging out that they have been created). The Promise callback is passed a "UserCredential" object, which contains data about the sign up that just occurred. Most usefully, this object contains a `user` property that is a "Firebase User" object, which contains data about the user who has just been created and signed in. For example, the Firebase User object contains a `uid` property, which is the *unique id* of that user generated by Firebase. This is like the "internal codename" for users that you can use to identify them.

Pro-tip: You don't need to come up with real email addresses for testing. Try using `a@a.com`, `b@a.com`, `c@a.com`, etc. Similarly, `password` works fine for testing passwords (though you should never do that in real life!)

User Profiles

It is also possible to store some additional information for each user in what is called the user's **profile**. Specifically, each user can assigned a `displayName` (a username separate from an email address) and a `photoURL` (a link to a profile picture for that user).

```
let name = firebaseUser.displayName; //the user's name
let pic = firebaseUser.photoURL; //the user's picture
```

You can specify these profile properties of a Firebase User by by calling its `updateProfile()` method, passing it an object with the new values to assign to those properties:

```
firebaseUser.updateProfie({
    displayName: "Ada",
    photoURL: "http://domain.com/picture.png"
})
```

- This method *also* returns a Promise, so you can use `.then()` to do something after the profile is updated, or `.catch()` to handle any errors. Note that a good practice is to call this method from inside the `createUserWithEmailAndPassword()` callback (on the Firebase User found in the User Credentials), and then return the resulting promise for handling, thereby "chaining" the user creation and profile specification.

You **cannot** assign values directly to any Firebase variables (including Firebase Users), since that data needs to be uploaded to the web (via an AJAX request).

Instead, you will need to call a method to update these objects.

Authentication Events

The `firebase.auth()` variable will keep track of which Firebase User is currently logged in—and this information persists even after the browser is closed. This means that every time you reload the page, the `firebase.auth()` function will perform the authentication and "re-login" the user.

The recommended way to determine who is currently logged in is to register an **event listener** for to listen for events that occur with the "state" of the authentication changes (e.g., a user logs in or logs out). This event will occur when the page first loads and Firebase determines that a user has previously signed up (the "initial state" is set), or when a user logs out. You can register this listener by using the `onAuthStateChanged` method:

```
let authUnregFunc = firebase.auth().onAuthStateChanged((firebaseUser) => {
    if(firebaseUser){ //firebaseUser defined: is logged in
        console.log('logged in');
        //do something with firebaseUser (e.g. assign with this.setState())
    }
    else { //firebaseUser undefined: is not logged in
        console.log('logged out');
    }
});
```

- Because this authentication involves connecting to the Firebase app, it my perform network access. Thus in a React application, this listener should be registered in the **`componentDidMount()`** lifecycle callback.

- The `onAuthStateChanged()` method takes a callback as a parameter, which will be handed a value representing the current Firebase User (or undefined if no one is logged in).

  The most common practice is to then take this passed in object and assign it to a more global variable, such as a `state` variable in a React function (e.g., `this.setState({currentUser: firebaseUser})`).

- The `onAuthStateChanged()` method returns a *new function* that can be used to "unregister" the listener when you want to stop responding to authentication changes. You can save this variable for later (e.g., as an instance variable).

  In React applications, you will want to unregister the listener when the component is removed, in the **`componentWillUnmount`** lifecycle callback.

Signing In and Out

You can sign a user in or out by using addition methods called on `firebase.auth()`, which do exactly what they suggest:

```
//sign in a user
firebase.auth().signInWithEmailAndPassword(email, password)
   .catch(err => console.log(err)); //log any errors for debugging

//sign out a user
firebase.auth().signOut()
    .catch(err => console.log(err)); //log any errors for debugging
```

- Both of these methods return Promises, so you can `.catch()` and display any errors.

20.3 Realtime Database

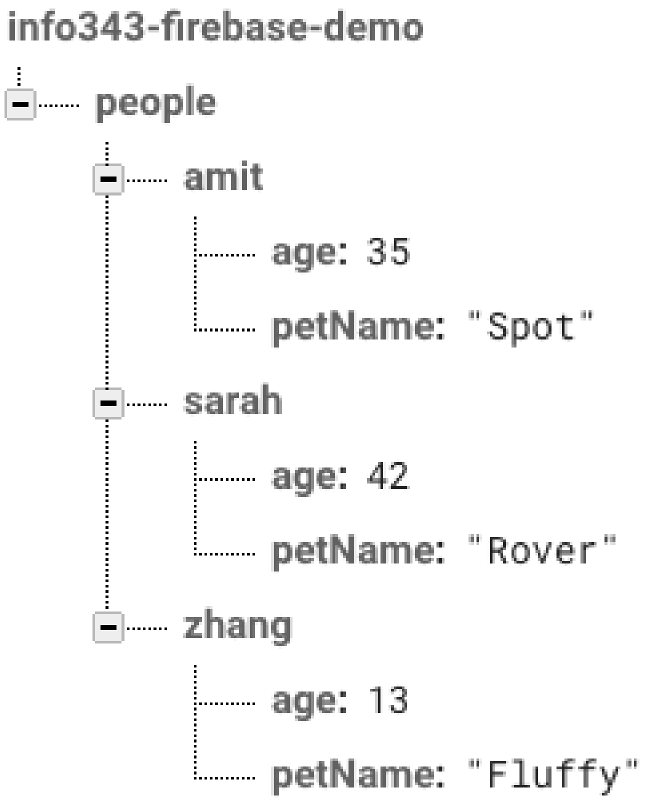
One of the main features provided by the Firebase web service is a **realtime database** for storing and accessing data in the cloud. You can think of this database as being a *single giant JSON object in the Cloud* that can be simultaneously accessed and modified by multiple clients—and since each client reads the same database, changes made by one user will be seen by others as *real-time updates*.

For example, you might have a database that is structured like:

```
{
  "people" : {
    "amit" : {
      "age" : 35,
      "petName" : "Spot"
    },
    "sarah" : {
      "age" : 42,
      "petName" : "Rover"
    },
    "zhang" : {
      "age" : 13,
      "petName" : "Fluffy"
    }
  }
}
```

This database object has a `people` key that refers to an object, which itself contains keys that refer to individual "person" objects (each of which has an `age` and `petName` property).

In the Firebase Web Console (under the "Database" tab in the navigation menu), this data structure would be presented as:



```
info343-firebase-demo
  people
    amit
      age: 35
      petName: "Spot"
    sarah
      age: 42
      petName: "Rover"
    zhang
      age: 13
      petName: "Fluffy"
```

Figure 20.3: A Firebase database.

- Note that in the Firebase Web Console you can edit this database directly: viewing, adding, editing, and deleting elements in the JSON. This is useful for debugging—both to check that your code is editing the database accurately, and to clean up any mistakes.

Although the JSON database can have many levels of nested objects, best practice is to try and keep the structure as "flat" as possible. This avoids you needing to download the nested "details" for a value if you only need to know e.g., the key names. See Structure Your Data for more details and examples.

Security Rules

Because the Firebase database is just a giant JSON object in the cloud and is usable from a client-side system, technically *anyone* can access it. Each element of the JSON object is accessible via AJAX requests (which are sent via `firebase` functions).

In order to restrict what clients and access this information (read or write to values in the JSON), Firebase lets you define security rules that specify what users can access which elements. For example, you can make it so that only *authenticated users* can add new entries the database, or that only specific users can update specific entries (e.g., the comments they wrote on a blog).

- By default, the database can only be accessed and modified by *authenticated users*. Thus you almost always need to modify the rules to allow people to utilize your website without being logged in.

In order to set up the security rules, you need to click on the "Database" link in the side navigation of the Firebase Web Console. Under the "Rules" tab, you can see the default defined rules defined for the database. Firebase Security Rules are defined in JSON using a very particular and difficult to understand schema (particularly if you want to customize the access to particular values in the database)

- Basically, inside the "rules" object you specify a JSON tree that mirrors the structure of your database. But instead of having a value for the keys in your database, you specify an object with `".read"` and `".write"` properties. The values of these properties are boolean expressions that state whether or not the current user is able to read (access) or write (modify) that value.

Luckily, Firebase provides a handy set of **sample rules** that cover the most common situations. For example:

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

This specifies that everyone can both read and write the entire database. This is a good rule to begin with for testing. For more details on how to write specific security rules, see Secure Your Data

- Be sure and hit "Publish" to save your changes!

Reading and Writing Data

Once you've set up access rules for your database, you can begin reading and writing data to it programmatically from your JavaScript. To do this, you will need to first get a *reference to the database*. This is done using the `firebase.database().ref()` method. This method takes as a parameter the key-path of the element in the JSON you wish to access:

```
//get reference to the "root" of the database: the containing JSON
let rootRef = firebase.database().ref();

//refers to the 'people' value in the database
let peopleRef = firebase.database().ref('people');

//refers to the "sarah" value inside the "people" value in the database
//similar to `database.people.sarah` using dot notation
let sarahRef = firebase.database().ref('people/sarah');
```

Indeed, every single value in the Firebase database can be accessed using a URI-style "path notation", where each nested key is indicated by a slash `/` (rather than the `.` in dot notation). Since this parameter is just a string, you can use string concatenation to construct the path to a particular value you wish to reference (e.g., if you only want to b working with the value for a particular person).

- In fact, this "path" can be used to reference that database value as a specific *resource* (in the RESTFul sense). Each database entry can be accessed at `https://my-project-name.firebaseio.com/path/to/value`. This is how you can modify Firebase by using pure AJAX requests (instead of the `firebase` library).

Alternatively, you can use the `.child()` method to get a reference to a specific "child" element in the database:

```
//this is equivalent to the above
let sarahRef = firebase.database().ref('people').child('sarah');
```

Once you have a reference to a particular entry in the database, you can modify the value at that entry using the **`.set()`** method. This method takes as an argument an *object* containing the keys and values you wish to assign to the that particular reference:

```
let sarahRef = firebase.database().ref('people').child('sarah');

//change Sarah's age to 43 (happy birthday!)
sarahRef.set( {age: 43} )
    .catch(err => console.log(err)); //log any errors for debugging
```

- Similar to React's `setState()` method, the Firebase's `set()` method will *overwrite* the values for any keys specified in its parameter, but leave other values unchanged. Any keys that did not previous exist will be added:

```
sarahRef.set( {favFood: "pizza"} ); //add a new key

firebase.database().ref('people').set({ //add a new entry to `people`
        ada: {age:27, petName:"Charles"} //27-year-old Ada has pet named "Charle
    });
```

Listening for Data Changes

Because Firebase is structured as *realtime* database (that may change over time), you read data from it by registering an **event listener** to listen for changes to that database. Firebase provides a method `.on()` for registering such listeners. Similar to the DOM's `addEventListener()` function, Firebase's `on()` method takes two parameters: an event name (as a string) and a callback function that should be executed when the event occurs:

```
let amitRef = firebase.database().ref('people/amit');
amitRef.on('value', (snapshot) => {
    let amitValue = snapshot.val();
    console.log(amitValue); //=> { age: 35, petName: "Spot" }
    //can do something else with amitValue (e.g., assign with this.setState())
});
```

- While Firebase supports a number of different "database change events", the most common to listen for is the **`'value'`** event, which occurs when the data entry is first created *or* whenever it changes. Other events include `child_added`, `child_removed`, and `child_changed`, which can be useful if you want to know *how* the database changed (not just that it did change)!
- The callback function will be passed a data snapshot as a parameter. This is a wrapper around the database JSON tree (allowing you to navigate it e.g., with `.child()`). More commonly, you will want to convert it into an actual JavaScript object by calling the `.val()` method on it.
- Because this listener involves network access, in a React App you would want to register this listener in the `componentDidMount()` callback. You'll also need to "clean up" and remove the listener if the component gets removed (in the `componentWillUnmount()` callback) to avoid errors. You can remove a listener from a database reference by using the `.off()` method (to cancel the `.on()`):

  ```
  amitRef.off();
  ```

Note that you can also read a single value once (without needing to register and unregister a listener) by using the `.once()` method, which will return a promise that will contain the read data (once it is downloaded).

Firebase Arrays

When working with and storing data in the cloud, we often want to organize that data into lists using *arrays*. However, Firebase **does not** directly support support arrays: the JSON object in the sky only contains objects, not arrays! This is because Firebase needs to support **concurrent access**: multiple people need to be able to access the data at the same time. But since arrays are accessed by *index*, this can cause problems if two people try to modify the array at the same time.

The problem is that with an array, an index number doesn't always refer to the same element! For example, if you have an array `['a', 'b' 'c']`, then index `1` may initially refer to the `'b'`. However, if you add an element onto the beginning of the array, then suddenly that `1` index refers to the `'a'`, and so if a user was trying to modify the `'b'` before their machine was aware of the change, they may end up editing the wrong value! This bug is an example of a race condition, which can occur when two processes are modifying data *concurrently* (at the same time).

To avoid this problem, Firebase treats all data structures as Objects, so that each value in the JSON tree has a *unique* key. That way each client will always be modifying the value they expect. However, Firebase does offer a way that you can treat Objects as arrays: databases references support a **`push()`** method that will automatically add a value to an object with an **auto-generated key**.

```
let tasksRef = firebase.database().ref('tasks'); //an object of tasks
tasksRef.push({description:'First things first'}) //add one task
tasksRef.push({description:'Next things next'}) //add another task
```

This will produce a database with a structure:

```
{
  "tasks" : {
    "-KyxgJhKOVeAj2ibPxrO" : {
      "description" : "First things first"
    },
    "-KyxgMDJueu17348NxDF" : {
      "description" : "Next things next"
    }
  }
}
```

- Notice how the `tasks` is an Object (even though we "pushed" elements into it), and each "task" is assigned an auto-generated key. You would thus still be able to interact with `tasks` as if it were an array, but instead of using a value from `0` to `length` as the index, you'll use a generated "key" as the index.

Firebase *snapshots* do support a `forEach()` function that you can use to iterate through their elements, allowing you to loop through the elements in an array. However, if you want to do something more complex (like `map()`, `filter()`, or `reduce()`), you need an actual array. The best way to get this is use call `Objects.keys()` on the `snapshot.val()` in order to get an array of the keys, and then you can iterate/map that (accessing each element in the "array" using bracket notation).

Note that when looping through an "array", each element is treated handled separately from its key (the same way that a `forEach()` loop lets you work with array elements separately from their index). But since you *need* that key as an identifier in order to `ref()` and modify the JSON element later, you will need to make sure that you "save" the key in the object as you processing it:

```
//assume `tasksSapshot` is a snapshot of the `tasks` "array"
let tasksObject = tasksSnapshot.val(); //convert snapshot to value
let taskKeys = Object.keys(tasksObject);
let taskArray = taskKeys.map((key) => { //map array of keys into array of tasks
    let task = tasksObject[key]; //access element at that key
    task.key = key; //save the key for later referencing!
    return task; //the transformed object to store in the array
});
```

Don't lose your key!

This is a quick overview of some major functions provided by Firebase. The service also offers additional options, including cloud storage for larger media such as images or video. For details and additional function, see the official documentation.

Resources

- Firebase Web Guide
- Firebase Authentication
- Firebase Realtime Database Guide

Chapter 21

Interactive React (Revised)

This chapter how websites built using the React library can be made interactive, with Components that render different content in response to user actions. Specifically, it details how to idiomatically handle *events*, store dynamic information in a Component's *state*, and perform specific actions (such as downloading data) in conjunction with a Component's *lifecycle events*.

21.1 Handling Events in React

You can handle user interaction in React in the same way you would using the DOM or jQuery: you register an *event listener* and specify a *callback function* to execute when that event occurs.

In React, you register event listeners by specifying a React-specific attribute on an element. The attribute generally named with the word **`on`** followed by the name of the event you want to respond to in *camelCase* format. For example, `onClick` registers a listener for `click` events, `onMouseOver` for `mouseover` events, and so on. You can see the full list of supported event handling attributes in the documentation for synthetic events. The attribute should be assigned a value that is a reference to a *callback function* (specified as an inline expression).

```
//A component representing a button that logs a message when clicked
class MyButton extends Component {
  //method to call when clicked. The name is conventional, but arbitrary
  //the callback function will be passed the DOM event
  handleClick(event) {
     console.log('clicky clicky');
  }
  render() {
    //make a button with an `onClick` attribute!
    //this "registers" the listener and sets the callback
    return <button onClick={this.handleClick}>Click me!</button>;
  }
}
```

This component renders a `<button>` in the DOM with a registered click event. Clicking on that DOM element will cause the component's `handleClick` method to be executed. Note that the method is referenced using the `this` keyword (and no parentheses!), because the reference is to *this Component's* `handleClick()` method.

Importantly, you can only register events on React elements (HTML elements like `<button>`, that are named with lowercase letters), not on Components. If you tried to specify a `<MyButton onClick={callback}>`, you would be passing a *prop* that just happened to be called `onClick`, but otherwise has no special meaning!

Although functionally similar, React event handling attributes are **NOT** the same as HTML event handles—React's `onClick` is different from HTML's `onclick`. The React attributes are specialized to work within the React event system. In the end, they're similar to a JSX-based syntactic shortcut for using `addEventListener()`, so we aren't actually mixing concerns more!

21.1.1 Accessing `this` Component from Events

It is very common to have a Component's event callback functions need to reference the instance variables or methods of that component—such as to do something based with a prop (found in `this.props`) or to call an additional helper method (called as `this.otherMethod()`). In short—the event callback will need to have access to the `this` context.

But as discussed in Section 15.2, callback functions are not called on any particular object, and thus do not have a value assigned to their `this` variable. In the example above, although you're using the word `this` to *refer to* the method `this.handleClick` (to tell JavaScript where to find the function), that function is not actually being called on the class. As described in Section 15.2, just because the method was defined inside a class doesn't mean it needs to be called on an instance of that class!

```
//BUGGY CODE: A button that causes and error when clicked!
class MyButton extends Component {
  handleClick(event) {
     //Reference the object's `this.props` instance variable. But since `this`
     //is undefined when executed as a callback, it will cause a
     //TypeError: Cannot read property 'props' of undefined
     console.log("You clicked on", this.props.text)
  }

  render() {
    //Specifies function (which happens to be a class method) as a callback
    return <button onClick={this.handleClick}>{this.props.text}</button>;
  }
}

//Render the component, passing it a prop
ReactDOM.render(<MyButton text="Click me!"/>, document.getElementById('root'));
```

As such, you will need to make sure to "retain" the `this` context when specifying an event callback function. There are a few ways to do this.

- First, as described in Chapter 15, an *arrow function* will utilize the same lexical `this` as the context it is defined in. Thus you can "wrap" the event callback function in an arrow function in order to keep the `this`, calling that method on the `this` instance explicitly:

  ```
  class MyButton extends Component {
    handleClick(event) {
       console.log("You clicked on", this.props.text) //functions as expected!
    }

    render() {
      return (
          <button onClick={(evt) => this.handleClick(evt) }>
              {this.props.text}
          </button>
      )
    }
  }
  ```

  In this example, the `onClick` listener is passed an anonymous callback function (in the form of an arrow function), which does the work of calling the `handleClick()` method on the instance (`this`). In effect, you're defining a "temporary" recipe to register with the event listener, whose one instruction is "follow this other recipe". Notice that this approach also has the bonus feature of enabling you to pass additional arguments to the event callback!

  React does note that this approach can have a performance penalty—you are creating a new function (the arrow function) every time the component gets rendered, and components may be rendered *a lot* as you make React apps interactive! This won't be noticeable as you are just getting started, but can begin to make a difference for large-scale applications.

- An alternative approach is to use a **public class field** to define the method. This is an *experimental* JavaScript syntax—it is currently being considered for official inclusion in the JavaScript language. However, the Babel transpiler supports this syntax (transforming it into a bound class function), and that support is enabled in Create React App allowing you to use the syntax *for React apps*.

A *public class field* is a field (instance variable) that is assigned at value the "top level" of a class, rather than explicitly assigning to a property of `this` in the constructor.

```
//A class with a public class field
class Counter {
    x = 0; //assign the value here, not in the constructor

    increment() {
        this.x = this.x + 1; //can access the field as usual
    }
}

let counter = new Counter();
counter.increment();
console.log(counter.x); //outputs 1
```

But since you can assign any type of value to a field—including functions—you can use a public class field and define the event callback, using an arrow function to maintain the bound `this` context:

```
class MyButton extends Component {
  //define event callback as a public class field (using an arrow function)
  handleClick = (event) => {
     console.log("You clicked on", this.props.text) //functions as expected!
  }

  render() {
    return <button onClick={this.handleClick}>{this.props.text}</button>;
  }
}
```

Although somewhat more tricky to read and interpret (particularly if the callback takes no parameters), this approach allows you to specify a bound function (which will have the correct value for `this`), while still being able to reference the function directly when registering the event listener—without having to wrap it in a separate arrow function. At the time of writing, this approach is the "cool" way that callback functions are specified in React.

21.2 State

The previous section describes how to respond to user-generated events, but in order for the page to be interactive you need to be able to manipulate its rendered content when that event occurs. For example, you could click on a button and show many times it was pressed, select a table column to sort the data by that feature, or select a component in order to show an entirely different "page" of content (see Chapter 19).

To achieve these effects, a Component will need to keep track of its state or situation—the number on the counter, how the table is sorted, or which "page" to show. In React, a Component's **`state`** represents internal, dynamic information about how that Component should be rendered. The state contains data that *changes over time* (and *only* such information—if the value won't change for that instance of a Component, it shouldn't be part of the state!).

React Components must store their state in the `state` instance variable (accessed as **`this.state`**). Unlike props that are specified as inputs to the Component, the state must be initially assigned a value, which should be done in the Component's *constructor*:

```
//A button that tracks how many times it was clicked
class CountingButton extends Component {
  constructor(props) { //the constructor must take a `props` parameter
     super(props)      //the constructor must call superclass constructor

     //initialize the Component's state
     this.state = {
         count: 0 //a value contained in the state
     }
  }

  render() {
    //can _access_ values from the state in the `render()` function
    return <button>You clicked me {this.state.count} times</button>;
  }
}
```

Because Components *inherit* (extend) the `React.Component` class, their constructors must do the same work as the parent class (so that they can function in the same way). In particular, the constructor must take in a single parameter (representing the props that are passed into the Component). It must then immediately call the parent's version of the constructor (and pass in those props) using `super(props)`. This will cause the props to be setup correctly, so that you can use them as normal.

Inside the constructor, you *initialize* the `this.state` value (and this is usually all you do in the constructor!). The `this.state` value must be a JavaScript

Object which can store specific data—you can't make the state a String or a Number, but an object that can contain Strings and/or Numbers (with keys to label them).

You can access the values currently stored in the state through the `this.state` instance variable. You will usually do this in the `render()` (or in a helper method called by the `render()` function). If a value doesn't get used for rendering, it probably doesn't need to be part of the state!

21.2.1 State vs. Props

Importantly, a Component's *state* is different from its *props*. Although state and props make look similar (they are both instance variables you access from the `render()` function), they have very different roles. Many developers get the two confused—to the point that React has an FAQ entry about the difference!

The key difference between props and state is:

> `props` are for information that doesn't change from the Component's perspective, including "initial" data. `state` is for information that will change, usually due to user interaction.

Props are the "inputs" into a Component, the values that someone else tells the Component it should. *Props are immutable*—from the Component's perspective; they cannot be changed once set (though a parent could create a *different* version of the Component with different props).

State is "internal" to the Component, and is (only) for values that change over time. If the value doesn't change over the life of the Component (e.g., in response to a user event), it shouldn't be part of the state! To quote from the documentation:

> State is reserved only for interactivity, that is, data that changes over time.

For example, a `SortableTable` component might be used to render a list of data. But as the date in the list would come from elsewhere (e.g., the overall `App`) and wouldn't be changed *by the table*, it would be passed in as a `prop`. However, the *order* in which that data is displayed might change—thus you could save an `orderBy` value in the state, and then use that to organize what elements are returned by the `render()` function. The data itself doesn't change, but how it is rendered does!

It is possible to use the props to initialize the state (since props are the "initial values" for the Component), though the props would not be changed later:

```
//A component representing a count
class Counter extends Component {
    constructor(props) {
        super(props)

        this.state = {
            //set initial state value based on prop
            count: this.props.startAt
        }
    }
}

let counter = <Counter startAt={5} />
```

In this case the `state` keeps track of the count, though the prop specifies an initial number. The `this.state.count` variable will change in the future, but the `this.props.startAt` value will not.

It's important that you *only* use state to track values that will change over the life of the Component. If you're not sure if a value should be part of the state, consider the following:

1. Is the value passed in from a parent via props? If so, it probably isn't state.
2. Does the value remain unchanged over time? If so, it definitely isn't state.
3. Can you compute it based on any other state or props in your component? If so, it definitely isn't state.

The last of these rules is important. In general, you should keep the state as *minimal as possible*—meaning you want to have a little data as possible in the state as possible. This is to help avoid duplicating information (which can get out of sync), as well as to speed up how React will "re-render" Components when the state changes.

One of the most difficult parts of architecting a React application is figuring out what information should be stored in the props or state (and in which Components). We suggest you practice using simple examples, and think carefully about it as you build more robust applications.

21.2.2 Changing the State

Data is stored in the state so that can be changed over time. You can modify a Component's state by calling the **`setState()`** method on that Component. This method usually takes as a parameter an object that contains the new desired values for the state; this set of new values will be "merged" into the existing state, changing only the indicated values (other values will be left alone):

```
//An element that displays the time when asked
class Clock extends Component {
  constructor(props) {
     super(props)
```

```
    this.state = {
        currentTime: new Date(), //current time
        alarmSound: "annoying_buzz.mp3" //changeable alarm sound
    }
  }

  //callback function for the button (public class field)
  handleClick = (props) => {
    let stateChanges = {
         currentTime: new Date() //new value to save in the state
    };
    this.setState(stateChanges); //apply the state changes and re-render!
  }

  render() {
     return (
         <div>
           <button onClick={this.handleClick}>What time is it right now</button>
           <p>The time is {this.state.currentTime.toLocaleTimeString()}</p>
         </div>
     );
  }
}
```

The `setState()` method will “merge” the values of its parameter into the Component’s `state` field; in the above example, the `alarmSound` value will not be modified when the button is pressed; only the value for `currentTime` will be changed. If you want to change multiple values at the same time, you can include multiple keys in the parameter to `setState()`. Also note that this merging is “shallow”—if you wanted to change a state value that was an array (e.g., `this.state = { comments:[...] }`), you would need to set a brand new version of that array (that could be a modified version of the previous state; see below).

Importantly, you *must* use the `setState()` method to change the state; you *cannot* assign a new value to the `this.state` instance variable directly. This is because the React framework uses that method to not only adjust the instance variable, but also to cause the Component to “re-render”. When the state has finished being updated, React will re-render the Component (causing it’s `render()` method to be called again), and merging the updated rendering into the page’s DOM. React does this merging in a highly efficient manner, changing the elements that have actually updated—this is what makes React so effective for large scale systems.

Remember: calling `setState()` will cause the `render()` method to be called again, and it will access the updated `this.state` values!

Never call `setState()` directly from inside of `render()`! That will cause an infinite recursive loop. The `render()` method must remain "pure" with no side effects.

Moreover, the `setState()` method is *asynchronous*. Calling the method only sends a "request" to update the state; it doesn't happen immediately. This is because React will "batch" multiple requests to update the state of Components (and so to rerender them) together—that way if your app needs to make lots of small changes at the same time, React only needs to regenerate the DOM once, providing a significant performance boost.

```
//An Component with a callback that doesn't handle asynchronous state changes
class CounterWithError extends Component {
  constructor(props) {
     super(props)
     this.state = {
         count: 3 //initial value
     }
  }

  handleClick = () => {
     this.setState({count: 4}); //change `count` to 4
     console.log(this.state.count); //will output "3"; state has not changed yet!
  }

  //...
}
```

In this example, because `setState()` is asynchronous, you can't immediately access the updated state after calling the function. If you want to use that updated value, you need to do so in the `render()` method, which will be called again once the state has finished being updated.

Because `setState()` calls are asynchronous and may be batched, if you wish to update a state value based on the *current* state (e.g., to have a counter increase), you need to instead pass the `setState()` method a *callback function* as an argument (instead of an Object of new values). The callback function will be passed the "current" state (and props), and must `return` the Object that you wish to merge into the state:

```
//An example button click callback
class Counter extends Component {
  constructor(props) {
     super(props)
     this.state = { count: 0 } //initial value
  }

  handleClick = () => {
     //setState is passed an anonymous callback function
     this.setState((currentState, currentProps) => {
         //return the Object to "merge" into the state
         let stateChanges = {count: currentState.count + 1}; //increment count
         return stateChanges;
     })
  }

  //...
}
```

While trying to use `this.state` directly in a call to `setState()` will *sometimes* work, best practice is to instead use a callback function as above when the new state value depends on the old.

Lifting Up State

In React, a component state is purely *internal* to that component—neither the Component's parent element nor its children have access to (or are even aware of) that state. State represents only information about that particular component.

Sometimes a child component (i.e., a Component instantiated in the `render()` method) needs some data that is stored in the state—for example, a `BlogPost` Component might need a data value that is stored in the `Blog`'s `this.state.comments` array. Components can make information available to their children by passing that data in as a *prop*:

```
class Blog {
    constructor(props){
        super(props);
        this.state = {posts: [...]} //initialize the state
    }

    render() {
        return (
            <div>
                <h2>Most Recent Post</h2>

                {/* pass in values from state as a prop */}
                <BlogPost postText={this.state.posts[0]} />
            </div>
        )
    }
}
```

In this example, the `BlogPost` would be passed the individual "post text" data as a normal prop, without being aware that the data was actually stored in the `Blog`'s state. When that data changes (e.g., when a new post is added), the `render()` function will be called again and the `BlogPost` Component will be re-instantiated with a new value for its prop—again, without any knowledge that there was a change. From the perspective of the `BlogPost`, there has only ever been a single `postText` value.

While passing state data to a child (as a prop) is easy, communicating state data to a *parent* or *sibling* is more complicated. For example, you may want to have a `SearchForm` component that is able to search for data, but want to them have a sibling `ResultsList` component that is able to display the rendered results:

```
<App>
  <SearchForm /> {/* has the data */}
  <ResultsList /> {/* needs the data */}
</App>
```

When confronted with this problem, the best practice in React is to **lift the state up** to the the *closest common ancestor*. This means that instead of having the data be stored in the state of one of the children, you instead store the data in the state of the *parent*. The parent can then pass that information down to the children (as props) for them to use. This way the data always flows "down" the tree.

But with the state stored in the parent, the children elements (e.g., the `SearchForm`) may still need a way to interact with that parent and tell it to change its state. To do this, you can have the parent define a function that changes its state, and then pass the child that *callback function* as a `prop`. The child will then be able to execute that callback prop, thereby telling the parent to do something (such as update its state!)

- It's like the parents write down their instructions, and then hand them to the child to do later!

In the following example, the parent Component (`VotingApp`) store state data about how many times each button has been pressed, passing a callback function (`countClick`) to its child Components (`CandidateButton`). The `CandidateButton` will then execute that callback when they are clicked. Executing this function causes the `VotingApp` to update its state and re-render the buttons, which show updated text based on the props given to them (namely: who is winning!).

```
class VotingApp extends Component {
  constructor(props) {
    super(props);
    this.state = {
      counts: { red: 0, blue: 0 } //initialize state (counts for each color)
    }
  }

  //expects a "color" for which button was clicked
  countClick = (color) => {
    console.log(color);
    this.setState((currentState) => {
      currentState.counts[color]++; //increment that color's count
      return currentState; //new value to assign to state
    })
  }

  render() {
    //render based on current state
    let winner = "tie";
    if(this.state.counts.red > this.state.counts.blue) winner = "red"
    else if(this.state.counts.blue > this.state.counts.red) winner = "blue"

    return (
      <div>
        <p>Current winner is: {winner}</p>
        {/* Pass the callback to each button as a prop */}
        <CandidateButton color="red" winner={winner} callback={this.countClick} />
        <CandidateButton color="blue" winner={winner} callback={this.countClick} />
      </div>
    )
  }
}

class CandidateButton extends Component {
  handleClick = () => {
    //On click, execute the given callback function (passing in own name)
    this.props.callback(this.props.color)
  }

  render() {
    //render based on current props
    let label = "I am not winning";
    if(this.props.winner === this.props.color)
      label = "I am winning!"

    return (
      <button className={this.props.color} onClick={this.handleClick}>
        {label}
      </button>
    );
  }
}
```

As with many React systems, there are a lot of moving parts that fit together.

To understand how this example works, try "tracing" the code that occurs when the `CountingApp` is first rendered, and then what happens when a button is clicked. Remember that calling `setState()` on a Component will cause that Component to be re-rendered!

Finally, notice in particular that the `CandidateButton` class knows nothing about its parent or indeed any information about how it is used (or even if there are other `CandidateButton` instances). Instead, it simply renders itself based on its props, and executes whatever `callback` function it was given whenever it is clicked (without carrying about what that function does).

In summary, in order to make an interactive React application, perform the following steps:

1. Start with a "static" (non-interactive) version, with appropriate Components
2. Identify variables that will change so need to be stored in the `state`
3. Put `state` in the "lowest" common ancestor for Components that need it
4. Pass `state` information to child Components as `props`
5. Pass *callback functions* as `props` to child Components so they can modify the state.

21.2.3 Working with Forms

One of the most common reasons to track state in a React app is when developing forms that the user can fill out to submit information. Forms are a common structure to use when getting user input—whether it's a "search form" for browsing a data set, or a "login form" for allowing a user access to personalized data.

In normal HTML, form elements such as **`<input>`** keep track of their own "state". For example, whatever the user has typed into a text input will be stored in that input's `value` property:

```
//Select the <input type="text"> element
let textInput = document.querySelector('input[type="text"]');

//Event that occurs whenever the input is changed
textInput.addEventListener('change', (event) => {
    let input = event.target;
    console.log(input.value); //access that elements "state"
});
```

You will often want React to be able to respond to user data entered into a form—either because you want to send an AJAX request based on that data, or because you want to perform **form validation** and confirm that the user has entered appropriate input (e.g., that the password is at least 6 characters long).

But storing the user input in the `<input>` element's state can cause problem for React: React components won't know when to update when that changes, and updating a React component might cause the entire DOM tree to re-render (producing a new `<input>` element with a different `value` state).

Thus the recommended practice for working with forms in React is to use **controlled Components**. In this case, you define a Component that will track the state of what the user has typed into the `<input>`, and then will render an `<input>` with an appropriate `value` property. Rather than letting the `<input>` control its own state, the React component controls the state and then dictates to the `<input>` element what value it should be showing. It's like React is snatching the data from the `<input>` and then claiming credit for it.

```
class MyInput extends React.Component {
   constructor(props) {
      super(props)
      this.state = {value: ''} //track the input's value in the state
   }

   //respond to input changes
   handleChange = (event) => {
      //get the value that the <input> now has
      let newValue = event.target.value

      //store that new value in the state, rendering the Component
      this.setState({value: newValue});
   }

   render() {
      return (
         <div>
            {/* The input will be rendered with the React-controlled value */}
            <input type="text" onChange={this.handleChange} value={this.state.value}
            <p>You typed: {this.state.value}</p>
         </div>
      );
   }
}
```

The above is an example of a basic controlled Component. When the user enters a different value into the `<input>` element, the `handleChange()` callback is executed. This grabs that updated value and saves it in the Component's state. Updating the state then causes the `render()` function to execute again, which recreates a brand new version of the `<input>` element, but displaying the React-controlled `value`. This way whatever value the user has entered will always be part of a React Component's state, and so can be manipulated and interacted with in the same manner.

- And of course, a controlled form might render multiple `<input>` elements, tracking the values of each one in a separate value in the state. This also allows the inputs to easily interact with each other, such if you want to confirm that a password was entered correctly twice.
- Robust form validation is actually quite tricky in React (especially when compared to other frameworks such as Angular). Using an external library such as Formik can help with developing forms and handling all the edge cases.

21.3 The Component Lifecycle

A react component's state is initialized in the constructor (when the component is first instantiated), and then usually modified in response to user events (by calling the `setState()` method). But there are a number of other "events" that occur during the life of a Component—such as the "events" of when the Component is added to the DOM ("mounted") or removed from the DOM ("unmounted"). For example, you should only download data when there is a Component on the screen to display that data (after the Component has been added to the DOM), and to "clean up" any listeners or timers when the Component is removed. It is possible to define functions will execute at such these events, allowing you to perform specific actions as the React framework manipulates the DOM. These functions are called **lifecycle methods**—they are methods that are executed at different stages of the Component's "lifecycle". You *override* these lifecycle methods in order to specify what code you want to run at those events. Lifecycle methods will be *automatically executed by the React framework*; you never directly call these methods (the same way you never directly call `render()`—which is itself a lifecycle method!)

React components have a number of different lifecycle methods, the most common of which are illustrated below:

```
//A generic component
class MyComponent extends Component {
  //The constructor is called when the Component is instantiated, but before_body
  //it is added to the DOM (on the screen)
  constructor(props){
    super(props)
    //initialize state here!
  }

  //This method is called when the Component has been added to the DOM (and
  //is visible on the screen). This occurs _after_ the first `render()` call.
  componentDidMount() {
    //do (asynchronous) setup work, including AJAX requests, here!
  }

  //This method is called when a Component is being "re-rendered" with a
  //new set of props. This is a less common method to override
  componentDidUpdate(prevProps, prevState, snapshot) {
    //do additional "re-setup" work (including updated AJAX requests) here!
  }

  //This method is called when the Component is about to be removed from the DOM
  //(and thus will no longer be visible on the screen)
  componentWillUnmount() {
    //do (asynchronous) cleanup work here!
  }
}
```

For more details on the specific usages (and parameters!) of these methods, see the official API documentation.

Note that you are not *required* to include these methods in a Component. However, they are required to correctly perform asynchronous functions such as AJAX requests, as described below.

21.3.1 Lifecycle Example: Fetching Data via AJAX

One of the most common use of lifecycle callback functions is when accessing data asynchronously, such as when fetching data via an AJAX request (such a described in Chapter 14). This section provides details about how to asynchronously load data within the React framework.

First, remember that React code is *transpiled* using Webpack. As such, some APIs—including `fetch()` are not "built-in" to React like they are with a modern browser. As discussed in Chapter 14, in order to support these "other" browsers, you will need to load a *polyfill*. You can do that with React by installing the `whatwg-fetch` library, and then `import`ing that polyfill in your React code:

```
# On command line, install the polyfill
npm install whatwg-fetch
```

```
//In your JavaScript, import the polyfill (loading it "globally")
//This will make the `fetch()` function available
import 'whatwg-fetch';
```

Remember that `fetch()` downloads data *asynchronously*. Thus if you want to download some data to display, it may take a while to arrive. You don't want React to have to "wait" for the data (since React is designed to be *fast*). Thus the best practice is to send the `fetch()` request for data, and `then` when the data has been downloaded, call the `setState()` method to update the

Component with the downloaded data. (The Component can initialize its state as an "empty array" of data).

Because `fetch()` will eventually call the `setState()` method, you can't send the AJAX from the Component's constructor. That's because `setState()` will eventually render the Component, which involves updating something that has been added to the DOM. In the constructor, the Component has been instantiated, but has not yet been added to the DOM—thus you can't update its state yet! If the data ends up downloading before the Component is mounted, you will get an error that you cannot re-render an unmounted Component!

Instead, you should always send your (initial) `fetch()` requests from the `componentDidMount()` lifecycle method. This way the data will only be downloaded once the Component has actually been added to the DOM, and so is available for re-rendering. This structure is shown in the example below:

```
class MyComponent extends Componet {
  constructor(props){
    super(props);
    this.state = {
        data: [] //initialize data as "empty"
    };
  }

  componentDidMount() {
    fetch(dataUri) //send AJAX request
      .then((res) => res.json())
      .then((data) => {
        let processedData = data.filter(...).map(...) //do desired processing
        this.setState({data: processedData}) //change the state, and re-render
      })
  }

  render() {
    //Map the data values into DOM elements
    //Note that this works even before data is loaded (when the array is empty!)
    let dataItems = this.state.data.map((item) => {
      return <li key={item.id}>{item.value}</li>; //return DOM version of datum
    })

    //render the data items (e.g., as a list)
    return <ul>{dataItems}</ul>;
  }
}
```

In the above example, the `this.state.data` is initialized as an empty array; this will `render()` just fine (it produces an empty list). Once the Component is mounted, the data will be downloaded and processed, and then saved as an updated state value. Calling `setState()` will cause the Component to re-render, so that the data will be displayed as desired! While technically it means the Component is rendering twice, React can batch these requests together so that if the data downloads fast enough, the user will not notice.

Resources

- Handling Events
- State and Lifecycle
- props vs state (blog)
- FAQ: Component State
- Lifting State Up
- React Forms
- The Component Lifecycle

Appendix A

Testing with Jest

> People just aren't as repeatable as computers are. Nor should we expect them to be. A shell script or batch file will execute the same instructions, in the same order, time after time. It can be put under source control, so you can examine changes to the procedure over time as well ("but it used to work…"). - The Pragmatic Programmer

This chapter introduces **automated testing** using the Jest framework. By following this tutorial, you will learn how to write and execute simple unit tests on JavaScript functions and DOM manipulating code.

This tutorial references code found at https://github.com/info343/jest-tutorial.

A.1 Testing

One of the most important goals when developing computer programs is to make sure that the code you write actually *works*. You can determine if a program works by considering three things:

1. What **input** was given to the program? From a user perspective, this is what *actions* that user took (e.g., "I pressed a button"). From a code perspective, this is often what value was passed to a method.
2. Running the program with a particular *input* will lead to a **received result**: something will happen because of the input. From a user perspective, this would be this would be like the changes to a web page caused by the button press. From a code perspective, this could be the value *returned* by that function (or potentially a new value for a *state* variable).
3. To know if a program worked though, you need to know the **expected result** of that program: what was *supposed* to happen because of the

input? If you don't know what the program was supposed to do, then you'll have no way of knowing if it worked or not!

You can test a program by providing the *input* and then comparing the *received result*. If the *received result* matches the *expected result*, then you know that the program worked! **Testing** is simply the process of providing the inputs and comparing the *received* and *expected* results

Providing the input and then comparing the received and expected results can get tedious, particularly if you have lots of possible inputs or they require multiple steps (like clicking on multiple buttons). For that reason, it's helpful to use **automation** to let a computer perform the testing for you. You define the expected result of some action, and the computer will check if the received result matches.

Because automated testing is so useful and comment, there are a wide variety of **testing frameworks**: external scripts that provide the code to let you easily compare received and expected results.

In JavaScript, the most popular testing frameworks are Jasmine, Mocha (which is just the framework, it uses Chai to actually compare received and expected results—hot beverages are a theme), and Jest. The later was developed by Facebook specifically to support testing React applications, and is the framework introduced and used in this course (though its API is almost identical to Jasmine and Mocha/Chai).

A.2 Testing with Jest

Jest is a command line program so you will need to have it installed on your machine. It can be installed *globally* through `npm`:

```
npm install -g jest
```

Additionally, you will need to install the dependencies listed in this repository's `package.json` file, which will allow tests to use ES6 module syntax to access the functions in a separate file to test (as well as auto-complete definitions for Jest!)

```
# install all dependencies
npm install
```

With Jest, you define "tests", which are just JavaScript code that is used to perform the *action* and compare the *received* and *expected* results. These tests can either be placed in a file whose name ends with `.test.js` (indicating it is a test script), `.spec.js` (for "specification"), or a plain `.js` file inside the `__tests__` folder in your program. In either case, a test script is *just a JavaScript file*, so you include any JavaScript code/variables/functions/etc. you want.

A.3 Writing a Test

We define a "test" (a check if a particular piece of functionality works) by calling the **`test()`** function—a predefined function provided by the Jest framework. This functions take two arguments: a *string* describing what the test is checking, and a *callback function* that will contain the code to run which does the action and compares the results.

```
test('should do something...', function() {
  //regular old Javascript code that will perform the test goes here
});
```

- Test descriptions (the string parameter) are written in *present tense*, and state in plain English what behavior SHOULD happen. Starting the description with the word "should" is a good approach!
- Jest also provides a function `it()` that is an alias for `test()`, and lets the code read like badly punctuated English:

  ```
  it('should do something', function() { ... });
  ```

Assertions and Matchers

We check that the program actually does what we're testing by writing an **assertion**. An assertion is a *proposition* that some fact is true. In this case, we are going to *assert* that the expected result and the actual result are the same. If that proposition is shown to be valid (our claim that the results matched is true), then we know that our claim that the program works must hold and thus we "pass" the test!

- You can think of an *assertion* as a bit like doing the work of an if-else statement:

  ```
  if(received value == expected value){
      test passes
  } else {
      test fails
  }
  ```

In Jest, we put forth an assertion by calling the **`expect()`** function with an appropriate **matcher**. The `expect()` function takes a single parameter, which is the *received result* (produced by doing the action; e.g., calling the function with a particular input). A matcher is *another function* that is called directly on the returned value of `expect()`, and takes as an argument the *expected result* that we want to compare. The matcher does the work of actually comparing the values, and then reporting the validity of our claim back to Jest:

```
test('should add numbers correctly', function() {
  expect(1+1).toEqual(2);
});
```

- In this case, `.toEqual()` is the "matcher" which compare the *received* and *expected* values to see if they are equal.
- Be careful about your parentheses! The `expect()` function should take a single expression (even if that expression includes a function call), and the matcher is called *after* the `expect()` function. Using local variables can help with readability.

Jest supports a wide variety of matchers. For example:

```
//can do numerical comparison
expect(receivedNumber).toBeGreaterThan(expectedNumber);

//compare against undefined
expect(receivedValue).toBeDefined(); //check if defined!

//find receivedValue in an array
expect(receivedArray).toContain(expectedValue);

//can negate ANY matcher with a .not property
expect(receivedValue).not.toEqual(expectedValue);
```

- See the documentation for a complete list.

Note that a single `test()` can include multiple assertions!

Organizing Tests

You can "group" tests together by using the `describe()` function. This function takes two parameters: a *string* that describes what **feature** is being tested, and a *callback function* that contains the code for the tests:

```
describe('Basic math', function() {
  it('should add numbers correctly', function() {
    expect(1+1).toEqual(2);
  });
});
```

- You should name your feature and test features so that they read as:

  {Feature name} should {do something specific}

  This will allow the test results to be communicated very clearly, even to non-developers (e.g., to your boss or client).

- `describe()` blocks can contain multiple `tests()`, and even other `describe()` blocks if you want to separate subfeatures.

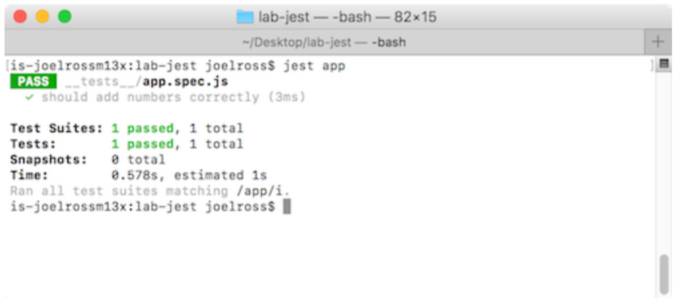
It is also possible to run particular blocks of code *before* groups of tests are run by using the `beforeEach()` and `beforeAll()` functions. See Setup and Teardown for details.

Running the Tests

You can **run** the test (have the computer perform the testing work) by using the `jest` command line program, passing it the name of the text script (without the extension) that you wish to run:

```
# test the app.spec.js file
jest app
```

The command line will print out the results of this test script:



```
is-joelrossm13x:lab-jest joelross$ jest app
 PASS  __tests__/app.spec.js
  ✓ should add numbers correctly (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.578s, estimated 1s
Ran all test suites matching /app/i.
is-joelrossm13x:lab-jest joelross$
```

Figure A.1: An example passing script

This will tell you what tests were run, which "passed" (were green for good to *go*!) and which "failed" (were red). Failed tests will also report which assertion failed, and what the expected and received values were that didn't match.

Practice

This repo's `app.js` file contains a function `invertCase()`. In the provides `app.spec.js` file, implement a `describe()` block to contains tests for the function, and write unit tests to check it for bugs (by giving it specific inputs and checking the *received* vs. the *expected* result). If you find any bugs, fix them and then re-run your tests!

A.4 Testing Web Apps with Jest

It is also possible to use Jest to JavaScript code that is used to manipulate the DOM. You can do this because Jest creates a **virtual DOM** that you can interact with. That is, jest provides a global `document` object that you can access and call methods on (e.g., `document.querySelector()`). This isn't a full browser: it won't load external CSS files or allow you to navigate to pages, but it does provide a tree of HTML elements you can modify and inspect. allowing you to test your DOM manipulation.

There are a few steps to being able to work effectively with the `document` provided by Jest:

1. First, you will want to make sure the content of the DOM includes the HTML elements you wish to test with. This is called **mounting** the content. You can mount some HTML content by assigning it to the `innerHTML` of the DOM's *root element*—not the `<html>` element, but a virtual element that acts as the ultimate parent of the DOM (similar to the root `/` folder on an operating system). This node can be accessed via the `document.documentElement` property:

   ```
   //assign a given HTML content (e.g., as string) to the virtual DOM
   document.documentElement.innerHTML = "<html><head></head><body>...</body></html
   ```

   Often, this HTML content is read directly from the file, using Node's `fs` (file system) library.

2. Because the `document` object only represents the DOM tree (the rendered HTML), it won't apply any embedded `<script>` tags. So in order to run your script on the DOM and modify it, you will need to *manually* apply those scripts. But since Jest has already defined you a `document` object to modify, you can simply tell Jest to load and run your script! Your code will do the exact same thing it does in the browser, just querying and modifying Jest's virtual DOM (`document`) rather than the browser's DOM.

   You load an external script in Jest by using Node's `require()` function, passing it the *relative path* to the script file you wish to load (this script must be saved locally):

   ```
   //load the `index.js` file
   require('../js/index.js');
   ```

   Note that this path is *relative* to the location of the test script, not relative to the location of the `.html` file (since Jest doesn't use the HTML). If the script file is in the same directory as the spec file, you will need to put a `./` in front of the filename path to indicate that you're giving a path rather than naming a module.

You will need to also explicitly have Jest load any external libraries (such as jQuery) you wish to use. These libraries will need to be loaded from a local file (Jest can't access a CDN), but they can usually be installed via `npm`. For example, you would load jQuery with:

```
$ = require('jquery'); //load the jQuery module (installed from npm)
window.$ = $; //assign make the jQuery library into a DOM global
```

*Remember to load any external scripts **before** your own!*

3. Finally, you can use DOM methods (or jQuery helpers!) to trigger *events* (like button clicks). You can then inspect the DOM with `document.querySelector()` and run assertions about the state of DOM after that user action.

```
//for example
let h1 = document.querySelector('h1');
expert(h1.textContent).toEqual('Hello world!');
```

And with that, you can automatically test if your page's interactivity works as expected without needing to repeatedly click on a button. In particular, these automatic tests can help you make sure that future changes don't *break* your code (e.g., don't cause the tests to fail), performing what is called **regression testing**.

- You should still test your page inside the browser, just to catch any platform differences between Jest's virtual DOM and actual web browsers.

Practice

Write a test in the provided `index.spec.js` file to confirm that when the "Panic" button is pressed, the HTML's `.alert` element is displayed.

Appendix B

Lab: Webpack

This lab is intended to walk you through the basics of the **Webpack** module bundling system. Webpack is the tool most commonly used in the React & Javascript community to *transpile* and *bundle* components, and is what is used under the hood by create-react-app (we'll be using this next week!). While that scaffolding tool means you don't need to know how to set up Webpack, it is good to be at least somewhat familiar with the concept (and this exercise will give you further practice working with `npm` modules).

This tutorial is adapted from one by Tyler McGinnis.

B.1 What is Webpack?

Webpack is a **build tool**. That is, it is a (command line) application that is used to automatically take the source code you write and prepare that code to be run/executed, whether for development or deployment. There are numerous such build tools in existence: Gulp is the major competitor to Webpack, and most IDEs (like jGrasp or IntelliJ) provide them. However, Webpack is favored by React developers because of how easily and speedily it transpiles JSX.

At its core, Webpack is a module loader: it takes source files like JavaScript **modules** and bundles them into a few simplified files that can be part of your webpage. It takes your complicated source code structure (lots of files) and transforms it into a brand-new, "simplified" version (few files).

B.2 Getting Started

There are two things you need to do to get started:

1. First, we need to create the `package.json` file to store information about the app you're building, including dependencies that Webpack will use. You can do this with the following command:

```
cd path/to/project
npm init
```

You will be prompted for a bunch of information to provide about your app. Give it the following details (**just hit `<enter>` to accept the details on any other prompts**)

- `name` should be "webpack-tutorial"
- `author` should be your name

Once you're finished, you will be asked to confirm your choices (type `yes`), and you'll have a brand new `package.json` file ready to use!

> :bulb: If you just want a blank `package.json` file, use `npm init -y` to skip the prompts!

2. Locally install the `webpack` program. This will allow you to run the program from the command line and bundle your app. Using the `--save-dev` flag saves the module to your `devDependencies`, which will not be bundled in the bundled version of your app.

```
npm install --save-dev webpack
```

B.3 webpack.config.js

While Webpack can be used to do simple bundling from the command line (see the official tutorial for an example), it's most common to write down all of your bundling options inside a **configuration file**. This file is basically just a JavaScript file that defines a variable that represents all of the different options you'd want to pass to the `webpack` program. If you name this file **`webpack.config.js`**, Webpack will read in that configuration automatically. Thus "using Webpack" really involves creating this file.

1. Create a new file called **`webpack.config.js`** and open it up in your favorite editor (e.g., VS Code).

2. Inside this file, add the following line of code:

```
module.exports = {}
```

This is the CommonJS version of the ES6 `export default {}`—you are defining an object (initially empty) that will be exported and used by the Webpack program. **The rest of this tutorial will involve adding properties to this object**.

- Note that is it *possible* to use ES 6 style `import` and `export` commands, but you need to name your file `webpack.config.babel.js` and will need to have some Babel libraries installed. See here for discussion. For this lab, stick to the CommonJS syntax.

Webpack's basic job is to take your source files, changing them in some way, and producing a new version. Thus there are three things you'll need to specify:

1. What files to transform (specifically: what JavaScript file makes your program start)
2. What transformations to make
3. Where to save the transformed files

entry and output

We specify the first piece (what files to transform) by giving the exported object an `entry` property:

```
module.exports = {
  //list of entry points
  entry: [
    __dirname + '/src/index.js'
  ]
}
```

This indicates which file (relative to the *folder you are currently in*) is the "start" or "entry point" into your program—in a way, which file has "main" in it. In our case, `src/index.js` (the `__dirname` is a Node constant referring to the current working directory). Note that the `entry` property is actually an *array*, since it's possible to support multiple entry points.

We'll skip step 2 for a moment and also specify where to save the transformed files. This is specified as the `output` property of the config object:

```
module.exports = {
  entry: [
    __dirname + '/src/index.js'
  ],
  output: {
    filename: "bundle.js",
    path: __dirname + '/dist'
  },
}
```

The `output` property is itself an object with more details about the output (rather than an array of possible outputs). The above example says that we should output into the `dist` folder (in the current directory), combining the code into a file called `bundle.js`.

Since we don't have webpack installed globally (which you're welcome to do if you wish `npm install -g webpack`), we'll need to add to our `package.json` file, to tell `npm` what to do with our local files!

Adding to your `package.json` file, we'll need to update the `scripts` key:

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack"
},
...
```

Now, if you run `npm run build`, the project will build! Similar to `npm install`, when we run this command, we tell `npm` that, within this project, we want to `run` the `build` script, which will call the local version of `webpack` to run. When `webpack` runs, it looks for the `webpack.config.js` file, and then executes using that file to build your project!

Here is roughly the output you should see:

```
> webpack-demo@1.0.0 build /Users/joelross/Desktop/webpack-demo
> webpack

Hash: fc5c075a9cfa7ee938c0
Version: webpack 3.8.1
Time: 56ms
    Asset  Size  Chunks             Chunk Names
bundle.js  3 kB       0  [emitted]  main
   [0] multi ./src/index.js 28 bytes {0} [built]
   [1] ./src/index.js 390 bytes {0} [built]
```

This will create a new file `dist/bundle.js`. If you view this file in VS Code, you'll see it contains some extra code that organizes the different **modules** into **functions** (to support variable scoping), one of which is the content of the `index.js` file!

B.4 Loaders

Look at all this red! This is the support that ES6 has among today's modern browsers. What does this mean? While ES6 features are being adopted more and more, there isn't great uniform support for it. So, we have to do a little trick called transpiling our code so that all browsers know what our Javascript is doing.

To do this, we use what are called **loaders**. A loader is basically a plugin that is used to perform a particular transformation (e.g., transpiling JSX or even ES 6 syntax!). Webpack's strength is its set of loaders that enable it to handle pretty much any kind of file and transformation. (It's of course possible to write your own, but that's well beyond the scope of this tutorial).

The `module` property is used to specify the list of `loaders`:

```
module.exports = {
  entry: [
    __dirname + '/src/index.js'
  ],
  output: {
    filename: "bundle.js",
    path: __dirname + '/dist'
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        loaders: ["babel-loader"],
      }
    ]
  },
}
```

The `module` property is an object that itself contains a `loaders` property, which is an *array* of loaders to apply. Each loader is described as an object (see? The nesting really does occur!).

- The `test` property indicates which file types we want the loader to transform. This may look scary and confusion, but that's because it's using a Regular Expression (similar to how you split words in the JavaScript Challenge) to specify the *file extension* that we want to consider. This particular expression indicates files that **end in `.js` or `.jsx`** (the later is often used for JSX React components). How to make sense of this expression and its crazy punctuation:
  - Regular expressions are like Strings, but surrounded by `/ /` instead of `" "`
  - The `.` in `.js` needs to be *escaped*, so has a `\` in front of it. Like `\n` for newlines.
  - The `x` in `.jsx` is *op tional* (without it we have `.js`, which is fine), so is written with a `?` after it to indicate that it can be present or not.
  - The last `$` indicates the "end of the line", so means we'll only talk about files that *end* in `.js` (e.g., `libray.js.css` wouldn't get transformed).
- the `loaders` property is a list of which loaders we want to apply to files whose names match the "test". In our case, we use Babel to transform our JSX, so we'll be using `"babel"` as our loader.

Babel Loader

Loaders such as Babel need to be installed individually using `npm` as if they were separate programs (since they are in fact separate libraries!). This we will need to install the `babel-loader` package to be able to apply Babel transformations:

```
npm install babel-loader --save-dev
```

- (The `--save-dev` argument here is like `--save` in that it saves the dependency into your `package.json` file. However, `--save-dev` lists the dependency as only needed for *development*, not for *deployment*. Thus if you wanted to upload your code to a web server (like on AWS), this would let that server know that it doesn't need to install Babel because you've already transpiled the code into a production build).

But because nothing is ever simple, the `babel-loader` actually requires an additional library (`babel-core`, which is the Babel program itself) to do its work. Thus we also need to install:

```
npm install babel-core --save-dev
```

Babel is able to perform all kinds of transformations, such as compiling JSX and converting ES6 syntax into older, browser-compatible versions. Babel is very modular, so each transformation we want to apply can be downloaded as an individual libraries called **presets**.

```
npm install babel-preset-es2015 --save-dev
```

This installs `babel-preset-es2015` (the transformation for ES 6).

Of course, downloading to preset transformations doesn't automatically tell Babel to use them. To do that, we actually need to create *another file* that will contain which presets Babel should use. This file is called **`.babelrc`** ("rc" stands for "run commands"; note the leading `.` indicating a hidden file). Create this file **in the same directory as `webpack.config.js`**.

Your `.babelrc` file just contains some JSON indicating which presets to use (other options are possible as well):

```
{
  "presets": ["es2015"]
}
```

Altogether, you've specified what Babel transformations to apply (in `.babelrc`), and told your Webpack config file to use Babel to modify any JavaScript files.

So finally, you should be able to use `webpack` to build a working version of your React program *next week*! Open the `index.html` file in a Browser to see your lovely app.

Some things to note:

- All your code is inside the `bundle.js` file.
- Whenever you change your code, you will need to "re-build" your application (run `webpack` again). There are further webpack plugins that can help automate this, such as ones that will automatically refresh the page when the source files change.

Clean up :bath:

Now that we've put in the hard work making webpack bundle our files, let's actually convert our files to proper ES6 synxtax and use correct import statements, given that only Chrome supports importing natively with special exceptions.

1. Be sure to export the functions from `src/sorter.js`, `src/looper.js` and `src/printer.js`!
2. Now that they're exported, use ES6 syntax in `src/index.js` to import the functions > Hint: 3rd syntax in the list here
3. Delete the `index.html` `<script>` tags for your files.
4. One last thing: in `src/sorter.js`, we're using an npm package called `lodash` to sort the array. Run `npm install --save lodash` (`--save` because we want the module to be used in our final app) and then import `lodash` in the `src/sorter.js` file. Hint: use the second import syntax from the MDN docs

With any luck, running `npm run build` one last time should give you a good output, and finally, opening `index.html` and looking at the console log should have your sorted user array!

Further Loader Practice

The webpage doesn't look great yet, because there is no styling (CSS) involved. To practice working with webpack loaders, add an `import` for `main.css` stylesheet to your `index.js`, and modify `webpack.config.js` so that it will bundle that file:

1. Add the import to your `index.js` file. The file path should be *relative* to the `index.js` file.

2. Install the **`style-loader`** and **`css-loader`** webpack loaders (remember to `--save-dev`). Together, these loaders are able to handle CSS files.

3. Add another element to the `module.loaders` array in the `webpack.config.js` file to specify the `style-loader` transformation.

   - The `test` should be a regular expression for files ending in **`.css`**
   - The `loader` itself should be `"style-loader!css-loader"`, which refers to the "css loader" module for the style-loader plugin.

4. Re-build your application using `webpack`. If you reload the page, you should now see it has a gray background!

There are lots of further configurations and options used by Webpack. For example, you can use the webpack-dev-server to have webpack run a local server that will automatically re-bundle modules when the files change. You are encouraged to check out that example if you have time.

Lab written in part by Evan Frawley

Appendix C

CSS in JS

This chapter discusses writing **CSS in JS**, a technique by which CSS styles and classes are defined in *JavaScript* rather than in separate CSS files. This technique is particularly common and useful in React, and solves a number of problems found in managing the styling of large web projects.

C.1 Why CSS in JS?

In November 2014, Facebook's Christopher Chedeau gave a **talk** outlining many of the issues that occur when trying to develop CSS for a large application:

A slide from Chedeau's talk. Click the image to view the entire deck.

In short, the talk points out how defining CSS classes in fact defines **global variables**: because stylesheets are loaded across the entire page, each class definition is effectively "global" and available to each Component within that page. If you define a class `.button` somewhere in your CSS, then every Component has access to the `.button` class.

- This becomes a problem when you want to support many different components. For example, you couldn't use `.button` for multiple buttons that are all styled very differently. Instead, you would need to define different classes for each button… and be careful to make sure that the names don't "conflict" (e.g., you don't try defining a `.button-submit` for two different kinds of submit buttons!). You can use careful naming schemes (such as BEM) to make sure that your names don't conflict, but this requires *a lot* of developer discipline and extra thinking—and one mistake can lead to difficult to track bugs!
- Moreover, CSS rules are global variables that include *implicit details* in their ordering (e.g., rules later in the stylesheet will override earlier ones).

> This is a problem when you're trying to load lots of different style sheets for different Components—which may be loaded in different orders or even asynchronously so you don't know which stylesheet will be loaded first!

The proposed solution to this problem is to define CSS style and class definition **in JavaScript as JavaScript variables**, and then using a library or build tool to convert those variables into properly name-spaced CSS classes that are included in the rendered DOM. This allow each Component to define its own styling (it's just a JavaScript variable) without worrying about conflicting global variables.

- Although the CSS and JavaScript will no longer be quite as separated, in the end the code you write will be simpler and easier to intuit. However, the DOM that these techniques produce will often look a lot messier (e.g., in the developer tools), with more complex class names or vast amounts of inline styling. This is generally an acceptable trade-off: the DOM isn't directly visible to most users, and is even ignored by screen readers.

Chedeau's talk and its solution were **hugely** influential, marking a major shift in how web developers thought about CSS. It led to the creation of a *large* number of projects that provide a different ways of including CSS in JS. These libraries all do mostly the same thing, but use different syntax to solve the problem of making it easier to develop non-global CSS styles.

C.2 React Inline Styles

The easiest way to include CSS in JS using React is to use its built-in support for inline styling. You can define a CSS rule as a JavaScript object:

```css
/* CSS version */
h1 {
  font-family: 'Helvetica';
  color: white;
  background-color: #333;
}
```

```js
/* JavaScript version */
const h1Style = {
  fontFamily: 'Helvetica',
  color: 'white',
  backgroundColor: '#333'
}
```

This coincidentally requires only a few changes from how you would normally write CSS properties: you need to *camelCase* property names like when you refer to DOM styling (since `-` isn't a legal character in JavaScript variables);

you need to put *all* property values in quotes, and you use `,` instead of `;` to separate properties in the object literal.

You can then apply this object to a particular element by specifying it as the **`style`** attribute using JSX:

```
<h1 style={h1Style}/>Hello World!</h1>
```

This is not actually defining a rule (it won't apply to *all* `<h1>` elements), but rather is using **inline styling** to apply the style to only a particular element.

Normally, in non-React contexts, inline styling (specifying CSS properties in an element's `style` attribute) is considered **bad practice**. It is difficult to modify and maintain, and leads to code duplication and poor cohesion (with style rules spread out across the program). This is part of why CSS-in-JS is considered "special": it takes what is usually bad practice and shows how, when used in a particular way (in React), it can actually produce *better* code!

You can even use JavaScript objects to *namespace* particular "style" objects, allowing you to produce something akin to CSS classes (where you can organize and apply lots of properties at once):

```
const styles = {
   success: {
      backgroundColor: 'green',
      color: 'white'
   },
   failure: {
      backgroundColor: 'red',
      color: 'white'
   }
}

//...
<button style={styles.success}>You win!</button>
<button style={styles.failure}>You lose.</button>
```

Note that these aren't real CSS classes, but rather are simply names given to styles that are applied *inline*. Thus you wouldn't refer to this button as `.success` (or even `.style.success`)—it is just a classless button that has some styles applied!

C.3 Aphrodite

React inline styles allow you to specify styling without creating global variables, but don't provide actual CSS classes. This means that you lose some semantic meaning (since you can't determine e.g., if a button is a "success" button just from the rendered DOM). You also lose the ability to handle more complex CSS rules, such as *pseudo-classes* (particularly ones like `:hover` or `:active`) and *media queries*.

For these features, you instead will need to use a **third-party library** for support CSS-in-JS. There are a wide variety of options for libraries that can be used for author robust CSS-in-JS. One of the cleanest (in the author's opinion) of these libraries is **Aphrodite**, which is developed and maintained by developers at Khan Academy. This library allows you to specify CSS classes as JavaScript objects, and then apply those classes to a React element via the `className` attribute as normal.

You specify a collection of style classes by using the `StyleSheet.create()` method supplied by the library. This method is passed an object whose keys are the "class names", similar to in the example above. You then reference the "classes" from this object by using the library's **`css()`** function:

```
import { StyleSheet, css } from 'aphrodite';

const styles = StyleSheet.create({
    success: {
       backgroundColor: 'green',
       color: 'white'
    },
    failure: {
       backgroundColor: 'red',
       color: 'white'
   }
});

//...
<button className={css(styles.success)}>You win!</button>
<button className={css(styles.failure)}>You lose.</button>
```

- Note that classes are still name-spaced (e.g., `styles.success`, not just `success`), you you pass the result of the `css()` function to the `className` property.

The Aphrodite library will take this `StyleSheet` you have defined and use it to **automatically generate CSS class rules** which are *injected* into the page, just as if you had to written them inside of a `.css` file! The above JSX code will produce DOM elements:

```
<button class="success_c72tod">You win!</button>
<button class="failure_cioc8l">You lose.</button>
```

Note that the class names start with the name you gave it, but have a number of additional characters at the end (the `_c72tod` after `success`). These characters are a hash of the CSS properties contained within that classname, and are used to distinguish between different classes after they have been injected into the page and thus have become "global" variables. In effect, this library will *automatically* produce "unique names" for each CSS class you define (differentiated by a deterministic hash at the end), so you don't need to worry about the `success` class in one Component interfering with the `success` class in another.

Aphrodite will also let you specify pseudo-selectors and media queries. These are specified as *properties* of the classes they should apply to; the name of that property is (*a string of*) the pseudo-selector or media query, and the value of that property is another object containing the CSS properties to apply:

```
const styles = StyleSheet.create({
    //...
    hover: {
        ':hover': {
            backgroundColor:'gray'
        }
    },
    responsive: {
        '@media (min-width:598px)': {
            fontSize:'2rem'
        }
    }
});
```

- This is different from how you normally define media queries: rather than specifying a media query with a block containing the classes it should apply, you specify a class whose block contains the media query.

You can pass multiple style classes (or arrays of classes) into the `css()` method, and they will automatically be combined into a *single unique style*:

```
//pass multiple styles into `css()`
<button className={css(styles.success, styles.hover)}/>You win!</button>
```

This will render an element with the DOM:

```
<button class="success_c72tod-o_O-hover_2nsohz">You win!</button>
```

Note here that the `success` and `hover` classes have been concatenated into *a single class* (using `-o_O-` as a separator); this is to avoid any "ordering" issues—ensuring that the `hover` options are always applied *after* the `success` options.

Aphrodite supports a few other features and edge cases, see the documentation for details.

Aphrodite is just one of many different CSS-in-JS libraries, all of which have their own syntax. But almost all of them will either generate an inline style to inject into an element, or will produce their own injected stylesheet with auto-generated class names. In either case, you can focus on just styling the components without worrying about the CSS stepping on its own toes!

C.4 CSS Modules

Another popular approach to solving the “global scoping” problem with CSS is to utilize **CSS Modules**. Rather than converting JavaScript objects into CSS styles and injecting them into the page, with CSS Modules you write your class definitions in `.css` files as normal. The CSS Modules then *post-processes* the `.css` files at build time (similar to how SASS works), converting the CSS classes into *locally scoped* versions. These locally scoped classes are then imported as JavaScript objects so that they can be referenced in a manner similar to React Inline Styles:

```css
/* app-styles.css */
.success {
   background-color: green;
   color: white;
}

.failure {
   background-color: red;
   color: white;
}
```

```
/* App.js */
import styles from './app-styles.css'

//...
<button className={styles.success}>You win!</button>
<button className={styles.failure}>You lose.</button>
```

This will render DOM elements with unique, locally-scoped class names (similar to what Aphrodite does):

```html
<button class="src-___App__success___1t37A">You win!</button>
<button class="src-___App__failure___2BnXi">You lose.</button>
```

Overall, CSS Modules has the advantages of letting you write styles in `.css` files as usual (it’s clear where your CSS is located), can run slightly faster (since in the end, it’s just a CSS file loaded as normal), and include a few extra features that supports easy **composition** of classes (similar to that provided by SASS).

Ejecting from Create React App

CSS Modules is a *post-processor* that "compiles" your CSS at build time, rather than at run time. Thus in order to utilize CSS Modules, you need to configure your application's "build system" to modularize the CSS. So far, you've utilized Create React App to provide a no-configuration build system (using webpack behind the scenes). However, Create React App doesn't support CSS modules by default—you need to modify the provided Webpack system to include the modularization step.

In order to modify the Webpack configuration build into Create React App, you will need to **eject** it. This process will change your project so that the Webpack configuration files are included as part of the source code, rather than loaded from a a single external library (`react-scripts`). Ejecting will allow you to modify how your React projects are built.

Ejecting is a one-way operation! Once you have "extracted" the build configurations, it's impossible to put them back (short of creating a new app and copying the files over). Be sure you want to do this! Note that if you *really* don't want to eject, there are other, more fragile solutions.

In order to *eject* your configuration information, run the following command:

```
cd path/to/app  # from the app folder
npm run eject
```

This will create two new folders in your project: `scripts` (which contains the `build` and `start` scripts, e.g., what happens when you run `npm start`), and `config/` (which contains the build configuration files).

- The Webpack configuration files specifically can be found at `config/webpack.config.dev.js` (configuration for the development server) and `config/webpack.config.prod.js` (configuration for production builds). In order to support CSS Modules, you will need to modify *both* the development and production configuration files.

You will need to make a simple change to the Webpack configuration files to support CSS Modules. In the `webpack.config.dev.js` file, at around line 160 (as of this writing), you will find a "rule" object with the property **`test: /\.css$/`**; this specifies what processing should be applied to `.css` files. Modify this property:

```
{
  test: /\.css$/,
  use: [
    require.resolve('style-loader'),
    {
      loader: require.resolve('css-loader'),
      options: {
        importLoaders: 1,
        //ADD THE BELOW TWO PROPERTIES!!
        modules: true,
        localIdentName: '[path]___[name]__[local]___[hash:base64:5]',
      },
    },
    //...
}
```

- These changes modify the already included `css-loader` loader so that it supports CSS Modules (a feature that is built into the loader, but not enabled by Create React App by default). The second line specify the "pattern" that should be used for generating the compiled class names: in this case, each class is named with the file path where it is used (imported), as well as its imported name.
- For `webpack.config.prod.js`, add the same `modules: true` property to the object dealing with `'css-loader'` (around line 180 as of this writing). You should not specify a custom `localIdentName`, since the default is a shorter hash that will run faster in production (though be less readable).

This is the only change you need to make to utilize CSS Modules! Now you can `import` CSS files into your React components as illustrated above, referring to each class as a *property* of the imported object which can be assigned to an element's `className` attribute!

- Note that if you want to apply multiple classes to an element, it's often easiest to utilize the `classnames` package, which gives you a helper method called `classNames` that will easily concatenate different classnames for you. See the examples for details.

react-css-modules

If needing to refer to classes as `style.classname` is tedious, you can use the `react-css-modules` library to simplify your React code. This library allows you to "decorate" each component with extra functionality—in particular, it allows you to use a **`styleName`** attribute to specify classes directly, without needing to namespace them:

```
import CSSModules from 'react-css-modules';
import styles from './app-styles.css';

class App extends Component {
    render() {
        return (
            <div>
                {/* Note the lack of `styles` namespaces! */}
                <button styleName={success}>You win!</button>
                <button styleName={failure}>You lose.</button>
            </div>
        );
    }
}

export default CSSModules(App, styles); //decorate the App so it reads from the styles
```

It is also possible to automate this decorating by using `babel-plugin-react-css-modules`, which is a *webpack plugin* that will automatically process the `styleName` attributes in the JSX *at build time* (e.g., it changes how the JSX is compiled!). This provides a significant performance benefit (as well as making your code cleaner).

- In order to support `babel-plugin-react-css-modules`, install the library with `npm` and modify the Webpack config file. Modify the "rule" object with the property **`test: /\.(js|jsx|mjs)$/`** (which applies to `.js` files):

  ```
  {
    test: /\.(js|jsx|mjs)$/,
    include: paths.appSrc,
    loader: require.resolve('babel-loader'),
    options: {
      cacheDirectory: true,
      //ADD THE BELOW PROPERTY!!
      plugins: [ 'react-css-modules' ]
    },
  }
  ```

  This will apply the plugin whenever the JavaScript files are transpiled, allowing you to utilize the `styleName` without explicitly using the `react-css-modules` library.

- Note that this plugin has a bug whereby changes to how CSS Modules are *composed* (see below) aren't applied when the ejected Webpack's developer server automatically reloads the browser window. See this issue.

Composing Classes

The other main feature of CSS Modules is the ability to **compose** classes—that is, you can specify that one CSS class *contains all the properties* defined by another. This is similar in functionality to the `@extends` keyword in SASS (though it only affects the classes as they are *exported* to JavaScript):

CSS classes are composed by specifying a **`composes`** property with a value that is the name of the class to "include":

```
.base {
  font-family: 'Helvetica';
  font-size: 2rem;
}
.success {
  composes: base; /* include the .base properties */
  background-color: green; /* additional properties only for .success */
}
.failure {
  composes: base;
  background-color: red; /* additional properties only for .failure */
}
```

When applied to the previous example (the two `<button>` elements), this will render as *two separate CSS classes*:

```
<button class="src-___App__success___1t37A src-___App__base___LeFt5">You win!</butto
<button class="src-___App__failure___2BnXi src-___App__base___LeFt5">You win!</butto
```

While you can still specify just a single class in your JavaScript (e.g., `className={style.success}`), CSS Modules will automatically apply all of the "dependent" styles to your element!

Moreover, you can also compose CSS classes **across separate files** by specifying the value as coming `from "filename"`:

```
/* colors.css */
.success {
    color: green;
}
```

```
/* app-styles.css */
.success {
    composes: base;
    composes: success from "../colors.css"; /* loads the color from another file */
}
```

This makes it possible to break up your CSS classes into a large number of individual "helper" modules: for example, you could have a `colors.css` file that defines coloring schemes, a `fonts.css` file that defines classes that only handle fonts (e.g., `.large`), a `layout.css` file that defines classes that only handle layout (e.g., `padding-small`, `margin-top-large`), and so forth:

```
/* example from docs */
.element {
  composes: large from "./fonts.css";
  composes: dark-text from "./colors.css";
  composes: padding-all-medium from "./layout.css";
  composes: subtle-shadow from "./effect.css";
}
```

In effect, you can use different files you develop your own set of Bootstrap-style utility classes!

- This may seem like overkill for a small app, but can be a great help when you're trying to design a large app (e.g., the size of Facebook) or a want to be able to consistently "theme" related but vastly different apps (think Gmail and Google Drive).

For more details and examples, see this tutorial introducing CSS Modules.

Resources

- React: CSS in JS - *the* talk pitching CSS in JS as an approach
- FAQ: Styling and CSS (React)
- Aphrodite documentation
- Inline CSS at Khan Academy - explanation and justification for the design choices used in Aphrodite.
- CSS Modules: Welcome to the Future - a step-by-step walk though and introduction

Appendix D

Redux

See official documentation at https://redux.js.org/, https://redux.js.org/docs/introduction/, and https://redux.js.org/docs/basics/.

Appendix E

React Native

As a special topic, this chapter provides a brief introduction to **React Native**, a framework for using React to build mobile applications (e.g., apps for Android or iOS). The React Native framework provides a set of build tools that allow you to *compile* React code (written in JavaScript) into *native* mobile code (Java for Android; Objective-C for iOS). This allows you to utilize your existing knowledge of client-side web development to also create mobile apps as if you had written them in their normal development language! Moreover, the same React code can be converted into apps for both iOS and Android (as well as for the web), with you only needing to adjust any platform-specific features. Three platforms for the price of one!

- This chapter's explanation mirrors the official tutorial for React Native; see that for more details.

E.1 Getting Setup

The easiest way to start a new React Native application is to use the **`create-react-native-app`** program. This works almost identically to the `create-react-app` program you know and love, except it will scaffold you a React Native application and provide build scripts utilizes for testing and developing native apps.

To use this program install it **globally**, and then execute it to create a new React Native project in the current folder:

```
create-react-native-app MyNativeApp
```

This scaffolding will include a number of configuration files (see the User Guide for details), but you'll mostly be interested in **`App.js`**, whose *default* export will be the "root" component of your application (mobile apps are usually designed around a View component that is shown, rather than an `index.js` style script to execute).

`create-react-native-app` doesn't currently work with `npm` version `5.0` or greater. Until this is fixed, the best solution is to use **yarn** to install and manage react native applications. Note that installing `yarn` may cause issues with `npm` that you'd need to resolve: proceed with caution if under a deadline! Alternatively, you can downgrade to an older version of `npm` using `npm install -g npm@4`

Running React Native Apps

There are a few different ways to "run" and test your application as you're developing it, depending on the platform (Android or iOS) you're targeting:

- For either platform, you can test your React Native code on a **physical device** by using the **Expo** app. This is a separate mobile application that "connects" to a server run by the `create-react-native-app` build scripts, displaying updates to your app in real time. In order to use *Expo*, install the app on your phone, and then use `npm start` to begin the development server. The server will show a QR code in the command line that you can scan to have your phone connect to the server (assuming they are on the same wireless network); this will run your developed app on the phone, and even automatically refresh it when you save changes to the file!

- It is also possible to run your React Native app on **virtual devices** (e.g., **emulators**) for either platform. These are "virtual" phones that run on your own computer, allowing you to develop and test mobile apps without needing a specific device.

  Note that you will need to have the appropriate development environment installed and set up for each platform: Android Studio for Android, and Xcode for iOS. Xcode only runs on MacOS.

  - For Android, you will need to create and start up an emulator with Android Studio: go to `Tools > Android > AVD Manager` to open up the Android Virtual Device Manager. You can then choose "Create Virtual Device…" in order to launch the wizard to specify a new emulator.

    You can then install and run an app on the emulator (using Expo) via the **`npm run android`** command. You can access the development menu on the device from the notification, or by hitting `cmd-m`.

  - For iOS, you can start up the Simulator program (it is found inside `Xcode.app/Contents/Developer/Applications`. If you right-click on the `Xcode.app` program and select "Show Package Contents", you will be able to navigate to it). You will then be able to install and run an app on Simulator (using Expo) via the **`npm run ios`** command. You can access the development menu on the device by hitting `cmd-d`.

- Finally, Expo Snack allows you to develop, test, and run simple React Native apps entirely online! This can be a good way to test out the design or to share code snippets with others.

E.2 React Native Apps

Writing React Native code uses the same process and techniques as writing normal React code: you define **Components** that `render` Views, which themselves are made up of more Components! These components can be passed in **props** and track **state** just like in React. In fact, if you look at the default **`App.js`** file created by `create-react-native-app`, you'll see that it's just basic React code.

However, instead of eventually rendering HTML elements (such as `<div>` or `<button>`), React Native apps render one of the framework's built-in components. Each of these components is able to be "compiled" into a an appropriate "native" version. For example, a `<View>` is compiled into a `<div>` element on the web, an `android.View` element on Android, and a `UIView` element on iOS.

- Just as React components almost always return a `<div>` with some content nested inside of it, React Native components almost always return a `<View>` with some content nested inside of it. `<View>` elements are particularly important when *styling* your app; see below.

Other basic components include:

- `<Text>` components represent displayed text (similar in purpose to a `<p>`, though they get compiled into inline `<span>` elements on the web). All displayed text must be inside one of these elements; you can't have a "text node" directly under a `<View>` like you can in HTML. `<Text>` elements also support cascading styling.

- `<Image>` components are used to display images (similar to a `<img>`). You specify which image to display by passing in a `source` prop. This can be a remote reference (e.g., `https://domain.com/picture.png`), or a local image. In order to refer to a local image file, you should import it using Node's `require()` method, specifying the path to the image file *relative to the Component*. This will the build tools to load the file as a data URI when rendering the image, and ensuring that the asset is packaged with the native app correctly.

  ```
  <Image source={ require('./path/to/picture.png') }/>
  ```

  Additionally, you should always specify an `accessibilityLabel` prop specifying how the image should be read to *screen readers* (yes, they exist for mobile devices! Blind people also use phones).

Styling React Native

You customize the appearance of React Native components by specifying styling properties in their **`style`** prop, similar to React inline styling. Note that Android and iOS don't support CSS, so you don't utilize `className` in React Native.

However, because it's often useful to organize styles into groups and give them labels, React Native provides a `Stylesheet.create()` similar to that used by Aphrodite. Defining a stylesheet helps with code organization, as well as providing a more efficient native implementation (since you don't need to duplicate style objects)—and efficient matters a lot more on resource-constrained mobile devices!

Although you specify style properties similar to CSS properties (e.g., with the same property names), React Native stylesheets do **not** use CSS! In particular, styles do not normally ***cascade***: specifying the `fontSize` for a `<View>` will not cause that property to be applied to multiple nested `<Text>` elements. This feature is missing because styling does not normally cascade in Android and iOS, and React Native can run more effectively by not needing to traverse the element tree to check if each and every property is defined by a parent. Moreover, this means that each component can be better *isolated* (developed as a stand-alone, drop-in piece of an application), because there is no chance of accidentally inheriting some styling.

- However, nested `<Text>` components will inherit from their parents as a convenience, allowing you to easily style parts of a text block (e.g., to make some text highlighted).

You can specify an element's size by setting it's `width` and `height` style properties. These properties should be assigned **unitless** numbers (you don't include `px` or `rem`). The value measures the number of *density-independent pixels*, which is a pixel-value that scales based on the resolution (dots-per-inch, or `dpi`) of the device. This allows the sizing to be consistent on "retina" displays.

Any elements that are not given a fixed size—as well as any element positioning—is primarily performed with Flexbox properties! The Flexbox framework allows you to provide a layout that will be consistent across different screen sizes. The "root" element (usually a `<View>`) of a component is rendered as a "flex item", but each `<View>` can also be made into its own "flex container" in order to specify the direction (which defaults to a vertical "column"), size, or spacing of its content:

```
//declare a stylesheet
const styles = StyleSheet.create({
    outer: {
        flex: 1; //this View should fill vertical space
        flexDirection: 'row'; //children should be layed out horizontally
    }
    inner: {
        flex: 1; //take up equal extra space
    }
})

//an example app
export default class App {
    render() {
        return (
            <View style={styles.outer}>
                <Text style={styles.inner}>Item 1</Text>
                <Text style={styles.inner}>Item 2</Text>
                <Text style={styles.inner}>Item 2</Text>
            </View>
        )
    }
}
```

Interaction

React Native apps can be made interactive using a similar process to regular React apps: you specify an event handler which can be used to modify the `state` and re-render the component. However, The events that you listen for are slightly different with React Native. For example, a `<Button>` element accepts an **`onPress`** property (instead of `onClick`):

```
<Button onPress={() => this.handlePress()}>Press me!</Button>
```

- Note that the callback is not passed any parameters, so you don't have an `event` to work with.

In order to get text input, you use a `<TextInput>` component (which is a lot like an `<input type="text">` element in HTML). This element can be made to be a *controlled input* just like with normal React, though you would use the `onChangeText` prop to listen for text changes (the callback function will be passed in the updated text):

```
<TextInput
    placeholder="Type something!"
    value={this.state.inputValue}
    onChangeText={ (newText) => this.setState({inputValue:newText}) }
    />
```

Lists and Data

Information applications often need to display *lists* of data values (e.g., a list of tasks to complete). While it is possible to `map()` an array variable to an array of `<View>` elements to render, Android and iOS support more specific techniques that allow for better responsiveness and efficiency when displayed on mobile devices. For example, these components will automatically render only a "portion" of the list that is currently visible on the small screen, loading new Views into memory only when the user scrolls down to see them. This allows the user to smoothly "flick" through a list of items.

You can create such an optimized list in React Native by rendering a `<FlatList>` component. This component takes two main properties: **`data`**, which is an array of data values to "map" into Views; and **`renderItem`**, which is a function that does the "mapping" (similar to the `render` function for `react-router`):

```
<FlatList
    data={myDataArray}
    renderItem={ (args) => <Text>{args.item.text}</Text> }
    />
```

- Each element in the `data` array *must* contain a `key` property that React uses to keep track of each item in the list (though you can pass a function that extracts a value as the key to the `keyExtractor` prop).

- The actual "data item" will be found in the `item` property of the callback function's arguments (the `index` property will contain the index of that item). However, it's common to use object destructuring to instead only pass the the specific property of the parameter object:

  ```
  function renderListItem({item}) { //param is `item = args.item`
      return <Text>{item.text}</Text>; //can access item directly
  }

  function renderWithIndex({item, index}) { //gets two params: args.item and args.
      return <Text>{index} - {item.text}</TEXT>;
  }
  ```

If you would like to download data from the internet to display using React Native, you use the `fetch()` API just like you've used in the web!

```
export default class App extends Component {
    constructor(props){
        super(props);
        this.state = {data:[]}
    }

    componentDidMount() {
```

```
        fetch(dataURI)
            .then((res) => res.json())
            .then((data) => {
                this.setState({data: data});
            })
            .catch((err) => console.error)
    }

    render() {
        return (
            <View>
                <FlatList data={this.state.data}
                    renderItem={({item}) => <Text>{item.text}</Text>}
                    />
            </View>
        )
    }
}
```

Overall, React Native is simply another way of *building* (but not implementing!) React applications, and makes heavy use of many of the modern frameworks and techniques (e.g., Flexbox, fetch) discussed throughout this course