Welcome to Web Programming Principles

The web is the most ubiquitous computing platform in the world. As a developer, learning the web takes time. There are hundreds of languages, libraries, frameworks, and tools to be learned, some old, some built yesterday, and all being mixed together at once.

The fundamental unit of the web is the hyperlink--the web is interconnected. These weekly notes provide numerous links to external resources, books, blogs, and sample code. To get good at the web, you need to be curious and you need to go exploring, you need to try things.

Make sure you follow the links below as you read, and begin to create your own web of knowledge and experience. No one resource can begin to cover the breadth and depth of web development.

> Question: do I need to read the weekly notes? How about all the many links to external resources?

Yes, you do need to read the weekly notes. You will be tested on this material. We will discuss it in class, but not cover everything. The external links will help you understand and master the material. You are advised to read some external material, but you don't need to read all of it. However, make sure you *do* read Recommended Readings.

Internet Architecture

Overview

- How does the Internet work?
  - How the Internet works in 5 minutes (video)
- How the Web works

Application Protocols

The web runs on-top of TCP/IP networks using a number of communication protocols, including:

- IP these 32-bit numbers (IPv4) are assigned to every device on the Internet (IPv6 uses 128-bit numbers).
- Domain Names human-readable addresses for servers on the Internet
- Domain Name System (DNS), the "Phone Book" of the Internet. There are many popular DNS servers you can use:
  - OpenDNS: `208.67.222.222`, `208.67.220.220`
  - Cloudeflare: `1.1.1.1`, `1.0.0.1`
  - Google: `8.8.8.8`, `8.8.4.4`
  - There are lots more, but each has trade offs (privacy, speed)
- Hypertext Transfer Protocol (HTTP)
  - How to get things on the web
  - HTTP Responses
- Hypertext Transfer Protocol Secure (HTTPS)

There are many more as well (SMTP, FTP, POP, IMAP, SSH, etc).

We often use the terms "Web" and "Internet" interchangeably, however, they aren't the same. Pictured below, Tim Berners-Lee (left), who invented the *World Wide Web*, and Vint Cert (right), who was one of the main inventors of the *Internet*:

Tim Berners-Lee (left) invented the web, and Vint Cert (right) invented the internet

The *World Wide Web* (WWW) runs on top of the Internet using HTTP, and allows us to access web services, request resources (i.e., pages, images), and transmit data between clients and servers. The web is a subset of the Internet.

The web isn't owned or controlled by any single company, organization, or government. Instead, it is defined as a set of open standards, which everyone building and using the web relies upon. Some examples of these standards include HTML, HTTP, SVG, and many more.

HTTP Requests and Responses

The Hypertext Transfer Protocol is a **stateless**, **client-server** model for formatting requests and responses between computers on the Internet. This means one computer makes a request (the client) to another (the server), and after the response is returned, the connection is closed.

The server listens for requests, and fulfills (or rejects) those requests by returning (or generating) the requested resources, such as images, web pages, videos, or other data.

URLs

Web resources are reachable via unique identifiers called a *Uniform Resource Locator* or *URL*. Consider the URL for this course's outline:

https://www.senecacollege.ca/cgi-bin/subject?s1=WEB222

A URL contains all the information necessary for a web client (e.g., a browser) to request the resource. In the URL given above we have:

- protocol: `https:` - the resource is available using the HTTPS (i.e., secure HTTP) protocol
- domain: `www.senecacollege.ca` - the domain (domain name) of the server. We could also have substituted the IP address (`23.208.15.99`), but it's easier to remember domain names.
- port: Not Given - if not specified, the port is the default for HTTP `80` or `443` for HTTPS. It could have been specified by appending `:443` like so: `https://www.senecacollege.ca:443`
- origin: combining the protocol, domain, and port gives us a unique origin, `https://www.senecacollege.ca`. Origins play a central role in the web's security model.
- path: `/cgi-bin/subject` - a filesystem-like path to the resource on the server. It may or may not end with a file extension (e.g., you might also have seen another server use `/cgi-bin/subject.html`)
- query string: `?s1=WEB222` - additional parameters sent to the server as part of the URL, of the form `name=value`

URLs can only contain a limited set of characters, and anything outside that set has to be *encoded*. This includes things like spaces, non-ASCII characters, Unicode, etc.

Requests

A URL describes the location (i.e., server, pathname) and how to interpret (i.e., which protocol) a resource on the Internet. To get the resource, we need to request it by sending a properly formatted HTTP Request to the appropriate server (host):

```
GET /cgi-bin/subject HTTP/1.1
Host: www.senecacollege.ca
```

Here we do a `GET` request using HTTP version 1.1 for the resource at the path `/cgi-bin/subject` on the server named `www.senecacollege.ca`.

There are various *HTTP Verbs* we can use other than `GET`, which allow us to request that resources be returned, created, deleted, updated, etc. The most common include:

- `GET` - retrieve the data at the given URL
- `POST` - create a new resource at the given URL based on the data sent along with the request in its *body*
- `PUT` - update an existing resource at the given URL with the data sent along with the request in its *body*
- `DELETE` - delete the resource at the given URL

We can use a URL in many ways, for example, via the command line using a tool like curl (NOTE: on Windows, use `curl.exe`):

```
$ curl https://www.senecacollege.ca/cgi-bin/subject?s1=WEB222

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
  "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html lang="en" dir="ltr"
  xmlns:content="http://purl.org/rss/1.0/modules/content/"
  xmlns:dc="http://purl.org/dc/terms/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:og="http://ogp.me/ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:sioc="http://rdfs.org/sioc/ns#"
  xmlns:sioct="http://rdfs.org/sioc/types#"
  ...
</section> <!-- /.block -->
  </div>
</footer>
</body>
</html>
```

Responses

Upon receiving a request for a URL, the server will respond with an *HTTP Response*, which includes information about the response, and possibly the resource being requested. Let's use `curl` again, but this time ask that it `--include` the response headers:

```
$ curl --include https://www.senecacollege.ca/cgi-bin/subject\?s1\=WEB222
HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Strict-Transport-Security: max-age=16070400; includeSubDomains
Date: Wed, 06 Sep 2023 14:31:11 GMT
Content-Length: 17241
Connection: keep-alive

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"><!-- InstanceBegin template="/Templates/mainTemplate.dwt"
codeOutsideHTMLIsLocked="false" -->
...
```

In this case, we see a two-part structure: first a set of **Response Headers**; then the actual HTML **Response Body**. The two are separated by a blank line. The headers provide extra metadata about the response, the resource being returned, the server, etc.

HTTP Headers are well defined, and easy to lookup via Google, MDN, or StackOverflow. They follow the `key: value` format, and can be upper- or lower-case:

`name: value`

For example, in the response above, we see a number of interesting things:

- `200 OK` - tells us that the requested resource was successful located and returned.
- Info about the `Date`
- The `Content-Type` is `text`, and more specifically, `html` (a web page) using ISO-8859-1 text encoding.

After these **headers** we have a blank line (i.e., `\n\n`), followed by the **body** of our response: the actual HTML document.

What if we requested a URL that we know doesn't exist?

```
$ curl --include https://ict.senecacollege.ca/course/web000

HTTP/1.1 302 Found
Date: Thu, 30 Aug 2018 20:25:28 GMT
Server: Apache/2.4.29 (Unix) OpenSSL/1.0.2l PHP/5.6.30
X-Powered-By: PHP/5.6.30
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, post-check=0, pre-check=0
Location: https://ict.senecacollege.ca/Course/CourseNotFound?=web000
Content-Length: 0
Content-Type: text/html; charset=UTF-8
```

This time, instead of a `200` status code, we get `302`. This indicates that the resource has moved, and later in the headers we are given a new `Location` to try. Notice there is no body (not every response will include one).

Let's try following the suggested redirect URL:

```
curl -I https://www.senecacollege.ca/cgi-bin/subjec
HTTP/1.1 404 Not Found
```

Now a third response code has been returned, `404 Not Found` as well as another HTML page telling us our course couldn't be located.

There are dozens of response codes, but they fall into a few categories you should learn:

- `1xx` - information responses
- `2xx` – successful responses
- `3xx` - redirection messages
- `4xx` – client error responses
- `5xx` – server error responses

Web Browsers

So far we've been communicating with web servers using `curl`, but a more common tool is a **Web Browser**.

A good way to think about a browser is as an operating system vs. an application. A web browser provides implementations of the web's open standards. This means it knows how to communicate HTTP, DNS and other protocols over the network in order to request resources via URLs. It also contains parsers for the web's programming languages, and knows how to render, execute, and lay-out web content for use by a user. Browsers also contain lots of security features, and allow users to download and run untrusted code (i.e., code from a random server on the Internet), without fear of infecting their computers.

Some of the the largest software companies and vendors in the world all have their own browsers:

- Google Chrome for desktop and Android
- Microsoft Edge and Internet Explorer (IE)
- Apple Safari and Safari for iOS
- Mozilla Firefox
- Samsung Internet for Android
- Opera

There are hundreds more, and thousands of different OS and version combinations. There are good stats on usage info for desktop and mobile, but no one company or browser controls the entire web.

As a web developer, you can't ever know for sure which browser your users will have. This means you have to test your web applications in different browsers and on different platforms in order to make sure the experience is good for as many people as possible.

The web is also constantly evolving, as new standards are written, APIs and features added to the web platform, and older technologies retired. A good way to stay on top of what does and doesn't work in a particular browser is to use https://caniuse.com/. This is a service that keeps track of web platform features, and which browsers do and don't implement it.

For example, you can look at the `URL()` API, used to work with URLs in JavaScript: https://caniuse.com/#feat=url. Notice that it's widely supported (green) in most browsers (89.69% at the time of writing), but not supported (red) in some older browsers like Internet Explorer.

Because the web is so big, so complicated, so old, and used by so many people for so many different and competing things, it's common for things to break, for there to be bugs, and for you to have to adapt your code to work in interesting ways. The good news is, it means there are lots of jobs for web developers to make sure it all keeps working.

Uniqueness of the Web as a Platform

We've been discussing HTTP as a way to request URLs be transferred between clients and servers. The web is globally distributed set of

- services - requesting *data* (Text, JSON, XML, binary, etc) to be used in code (vs. looked at by a user)
- resources, pages, documents, images, media - both static and dynamic user viewable resources (web pages), which link to other similar resources.
- applications - a combination of the above, providing rich user interfaces for working with real-time data or other complex information, alone or in networked (i.e., collaborative) ways.

The web can be read-only. The web can also be interactive (video games), editable (wikis), personal (blog), and productive (e-commerce).

The web is *linkable*, which makes it something that can be indexed, searched, navigated, and connected. The web gets more valuable as its connections grow: just look at all the other pages and resources this page links to itself!

The web allows users to access and run remote applications *without* needing to install new software. **The deployment model of the web is HTTP**. Compare that to traditional software that has to be manually installed on every computer that needs to run it. The same is true with mobile phones and apps in the various app stores. On the web, updates get *installed* every time you open a URL.

> Question: how many mobile or desktop apps did you install today vs. how many websites did you visit?

The web works on *every* computing platform. You can access and use the web on desktop and mobile computers, on TVs and smartwatches, on Windows and Mac, in e-Readers and video game consoles. The web works everywhere, and learning how to develop software for the web extends your reach into all those platforms.

Front-End Web Development: HTML5, CSS, JavaScript, and friends

When we talk about programming for the web in a browser, we often refer to this as *Front-End Web Development*. This is in contrast to server-side, or *Back-End Development*. In this course we will be focused on the front-end, leaving back-end for subsequent courses.

The modern web, and modern web browsers, are incredibly powerful. What was once possible only on native operating systems can now be done within browsers using only web technologies (cf. running Windows 2000 or Doom 3 in a browser window!)

The set of front-end technologies that make this possible, and are commonly referred to as the Web Platform, include:

- HTML5 - the Hypertext Markup Language, and its associated APIs, provide a way to define and structure content
- CSS - Cascading Style Sheets allow developers and designers to create beautiful and functional UIs for the web
- JS - JavaScript allows complex user interaction with web content, and dynamic behaviours in documents and applications.
- DOM - the Document Object Model and its APIs allows scripts and web content to interact at runtime.
- Web APIs - hundreds of APIs provide access to hardware devices, networking, files, 2D and 3D graphics, databases, and so much more.
- WebAssembly or WASM - a low-level assembly language that can be run in web browsers, allowing code written in C/C++ and other non-web languages to target the web. For example, Google Earth uses WebAssembly.

In addition to these primary technologies, an increasingly important set of secondary, or third-party technologies are also in play:

- Libraries, Modules - Bootstrap, Leaflet, Three.js, Lodash, ...
- Frameworks - React, Angular, Vue.js, ...
- Tooling - Babel, webpack, ESLint, Prettier
- Languages that "compile" to JavaScript - because JavaScript runs everywhere, many languages target the web by "compiling" (also know as *transpiling*) to JavaScript. A good example is TypeScript.

The front-end web stack is also increasingly being used to build software outside the browser, both on desktop and mobile using things like Electron and Progressive Web Apps (PWA). Visual Studio Code, for example, is written using web technologies and runs on Electron, which is one of the reasons it works across so many platforms. You can also run it entirely in the browser: vscode.dev.

Introduction to JavaScript

The first front-end web technology we will learn is JavaScript. JavaScript (often shortened to JS) is a lightweight, interpreted or JIT (i.e., Just In Time) compiled language meant to be embedded in host environments, for example, web browsers.

JavaScript looks similar to C/C++ or Java in some of its syntax, but is quite different in philosophy; it is more closely related to Scheme than C. For example, JavaScript is a dynamic scripting language supporting multiple programming styles, from object-oriented to imperative to functional.

JavaScript is one of, if not the most popular programming languages in the world, and has been for many years. Learning JavaScript well will be a tremendous asset to any software developer, since so much of the software we use is built using JS.

> JavaScript's many versions: JavaScript is an evolving language, and you'll hear it referred to by a number of names, including: ECMAScript (or ES), ES5, ES6, ES2015, ES2017, ..., ES2021, ES2022, etc. ECMA is the European Computer Manufacturers Association, which is the standards body responsible for the JS language. As the standard evolves, the specification goes through different versions, adding or changing features and syntax. In this course we will primarily focus on ECMAScript 6 (ES6) and newer versions, which all browsers support. We will also sometimes use new features of the language, which most browsers support. Language feature support across browsers is maintained in this table.

JavaScript Resources

Throughout the coming weeks, we'll make use of a number of important online resources. They are listed here so you can make yourself aware of them, and begin to explore on your own. All programmers, no matter how experienced, have to return to these documents on a routine basis, so it's good to know about them.

- JavaScript on MDN
  - JavaScript Guide
  - JavaScript Reference
- Eloquent JavaScript
- JavaScript for impatient programmers (ES2022 edition)

JavaScript Environments

Unlike C, which is compiled to machine code, JavaScript is meant to be run within a host environment. There are many possible environments, but we will focus on the following:

- Web Browsers, and their associated developer tools, primarily:
  - Chrome DevTools
  - Firefox Developer Tools
- node.js, and its command line REPL (Read-Eval-Print-Loop)

If you haven't done so already, you should install all of the above.

JavaScript Engines

JavaScript is parsed, executed, and managed (i.e., memory, garbage collection, etc) by an engine written in C/C++. There are a number of JavaScript engines available, the most common of which are:

- V8, maintained an used by Google in Chrome and in node.js
- SpiderMonkey, maintained and used by Mozilla in Firefox
- ChakraCore, maintained and used by Microsoft in Edge
- JavaScriptCore, maintained and used by Apple in Safari

These engines, much like car engines, are meant to be used within a larger context. We will encounter them indirectly via web browsers and in node.js.

It's not important to understand a lot about each of these engines at this point, other than to be aware that each has its own implementation of the ECMAScript standards, its own performance characteristics (i.e., some are faster at certain things), as well as its own set of bugs.

Running JavaScript Programs

JavaScript statements can be stored in an external file with a `.js` file extension, or embedded within HTML code via the HTML `<script>` element. As a developer, you also have a number of options for writing and executing JavaScript statements or files:

1. From the command line via node.js. You'll learn more about node.js in subsequent courses, but we'll also use it sometimes in this course to quickly try test JavaScript expressions, and to run JavaScript programs outside the browser.

2. Using Firefox's Developer Tools, and in particular the Web Console, JavaScript Debugger, and Scratchpad.

3. Using Chrome's DevTools, and in particular the Console and Sources Debugger

4. Finally, we'll eventually write JavaScript that connects with HTML and CSS to create dynamic web pages and applications.

Take some time to install and familiarize yourself with all of the methods listed above.

JavaScript Syntax

Recommend Readings

We will spend a month learning JavaScript, and there is no one best way to do it. The more you read and experiment the better. The following chapters/pages give a good overview:

- Chapter 1. Basic JavaScript of Exploring JS (ES5).
- MDN JavaScript Introduction Tutorial
- Chapter 1. Values, Types and Operators and Chapter 2. Program Structure of Eloquent JavaScript (2nd Ed.).

Important Ideas

- Like C, JavaScript is Case-Sensitive: `customerCount` is not the same thing as `CustomerCount` or `customercount`

- Name things using `camelCase` (first letter lowercase, subsequent words

start with uppercase) vs. `snake_case`.

- Semicolons are optional in JavaScript, but highly recommended. We'll expect you to use them in this course, and using them will make working in C++, Java, CSS, etc. much easier, since you have to use them there.

- Comments work like C/C++, and can be single or multi-line

```
// This is a single line comment. NOTE: the space between the //
and first letter.

/*
 This is a multi-line comment,
 and can be as long as you need.
 */
```

- Whitespace: JavaScript will mostly ignore whitespace (spaces, tabs, newlines). In this course we will expect you to use good indentation practices, and for your code to be clean and readable. Many web programmers use Prettier to automatically format their code, and we will too:

```
// This is poorly indented, and needs more whitespace
function add(a, b) {
  if (!b) {
    return a;
  } else {
    return a + b;
  }
}

// This is much more readable due to the use of whitespace
function add(a, b) {
```

- JavaScript statements: a JavaScript program typically consists of a series of statements. A statement is a single-line of instruction made up of objects, expressions, variables, and events/event handlers.

- Block statement: a block statement, or compound statement, is a group of statements that are treated as a single entity and are grouped within curly brackets `{...}`. Opening and closing braces need to work in pairs. For example, if you use the left brace `{` to indicate the start of a block, then you must use the right brace `}` to end it. The same matching pairs applies to single `'......'` and double `"......."` quotes to designate text strings.

- Functions are one of the primary building blocks of JavaScript. A function defines a subprogram that can be called by other parts of your code. JavaScript treats functions like other built-in types, and they can be stored in variables passed to functions, returned from functions or generated at run-time. Learning how to write code in terms of functions will be one of your primary goals as you get used to JavaScript.

- Variables are declared using the `let` keyword. You must use the `let` keyword to precede a variable name, but you do not need to provide a type, since the initial value will set the type.

> JavaScript version note: JavaScript also supports the `var` and `const` keywords for variable declaration. We will primarily use `let` in this course, but be aware of `var` and `const` as well, which other developers will use.

```
let year;
let seasonName = 'Fall';

// Referring to and using syntax:
year = 2023;
console.log(seasonName, year);
```

- JavaScript Variables: variables must start with a letter (`a-zA-z`), underscore (`_`), or dollar sign (`$`). They cannot be a reserved (key) word. Subsequent characters can be letters, numbers, underscores.

NOTE: If you forget to use the `let` keyword, JavaScript will still allow you to use a variable, and simply create a *global variable*. We often refer to this as "leaking a global," and it should always be avoided:

```
let a = 6; // GOOD: a is declared with let
b = 7; // BAD: b is used without declaration, and is now a global
```

- Data Types: JavaScript is a typeless language--you don't need to specify a type for your data (it will be inferred at runtime). However, internally, the following data types are used:
  - `Number` - a double-precision 64-bit floating point number. Using `Number` you can work with both Integers and Floats. There are also some special `Number` types, `Infinity` and `NaN`.
  - `BigInt` - a value that can be too large to be represented by a `Number` (larger than `Number. MAX_SAFE_INTEGER`,) can be represented by a `BigInt`. This can easily be done by appending `n` to the end of an integer value.
  - `String` - a sequence of Unicode characters. JavaScript supports both single (`'...'`) and double (`"..."`) quotes when defining a `String`.
  - `Boolean` - a value of `true` or `false`. We'll also see how JavaScript supports so-called *truthy* and *falsy* values that are not pure `Boolean`s.
  - `Object`, which includes `Function`, `Array`, `Date`, and many more. - JavaScript supports object-oriented programming, and uses objects and functions as first-class members of the language.
  - `Symbol` - a primitive type in JavaScript that represents a unique and anonymous value/identifier. They can normally be used as an object's

unique properties.

- `null` - a value that means "this is intentionally nothing" vs. `undefined`
- `undefined` - a special value that indicates a value has never been defined.

Declaration	Type	Value
`let s1 = "some text";`	`String`	`"some text"`
`let s2 = 'some text';`	`String`	`"some text"`
`let s3 = '172';`	`String`	`"172"`
`let s4 = '172' + 4;`	`String`	`"1724"` (concatenation vs. addition)
`let n1 = 172;`	`Number`	`172` (integer)
`let n2 = 172.45;`	`Number`	`172.45` (double-precision float)
`let n3 = 9007199254740993n;`	`BigInt`	`9007199254740993n` (integer)
`let b1 = true;`	`Boolean`	`true`
`let b2 = false;`	`Boolean`	`false`
`let b3 = !b2;`	`Boolean`	`true`
`let s = Symbol("Sym");`	`symbol`	`Symbol(Sym)`

Declaration	Type	Value
`let c;`	`undefined`	`undefined`
`let d = null;`	`object`	`null`

Consider a simple program from your C course, and how it would look in JavaScript

```c
 // Area of a Circle, based on https://scs.senecac.on.ca/~btp100/pages/content/input.html
 // area.c

 #include <stdio.h>                 // for printf

 int main(void)
 {
    const float pi = 3.14159f;   // pi is a constant float
    float radius = 4.2;          // radius is a float
    float area;                  // area is a float

    area = pi * radius * radius; // calculate area from radius

    printf("Area = %f\n", area); // copy area to standard output

    return 0;
}
```

Now the same program in JavaScript:

```javascript
const pi = 3.14159; // pi is a Number
let radius = 4.2; // radius is a Number
```

We could also have written it like this, using `Math.PI`, which we'll learn about later:

```
let radius = 4.2; // radius is a Number
let area = Math.PI * radius * radius; // calculate area from
radius

console.log('Area', area); // print area to the console
```

- Common JavaScript Operators (there are more, but these are a good start):

Operator	Operation	Example
`+`	Addition of `Number`s	`3 + 4`
`+`	Concatenation of `String`s	`"Hello " + "World"`
`-`	Subtraction of `Number`s	`x - y`
`*`	Multiplication of `Number`s	`3 * n`
`/`	Division of `Number`s	`2 / 4`
`%`	Modulo	`7 % 3` (gives 1 remainder)

Operator	Operation	Example
`++`	Post/Pre Increment	`x++`, `++x`
`--`	Post/Pre Decrement	`x--`, `--x`
`=`	Assignment	`a = 6`
`+=`	Assignment with addition	`a += 7` same as `a = a + 7`. Can be used to join `String`s too
`-=`	Assignment with subtraction	`a -= 7` same as `a = a - 7`
`*=`	Assignment with multiplication	`a *= 7` same as `a = a * 7`
`/=`	Assignment with division	`a /= 7` same as `a = a / 7`
`&&`	Logical `AND`	`if(x > 3 && x < 10)` both must be `true`
`()`	Call/Create	`()` invokes a function, `f()` means invoke/ call function stored in variable `f`
`||`	Logical `OR`	`if(x === 3 || x === 10)` only one must be `true`

Operator	Operation	Example
`|`	Bitwise `OR`	`3.1345|0` gives `3` as an integer
`!`	Logical `NOT`	`if(!(x === 2))` negates an expression
`==`	Equal	`1 == 1` but also `1 == "1"` due to type coercion
`===`	Strict Equal	`1 === 1` but `1 === "1"` is not `true` due to types. Prefer `===`
`!=`	Not Equal	`1 != 2`, with type coercion
`!==`	Strict Not Equal	`1 !== "1"`. Prefer `!==`
`>`	Greater Than	`7 > 3`
`>=`	Greater Than Or Equal	`7 >=7` and `7 >= 3`
`<`	Less Than	`3 < 10`
`<=`	Less Than Or Equal	`3 < 10` and `3 <=3`
`typeof`	Type Of	`typeof "Hello"` gives `'string'`, `typeof 6` gives `'number'`
`cond ? a`	Ternary	`status = (age >= 18) ? 'adult' :`

Operator	Operation	Example
`: b`		`'minor';`

> JavaScript version note: you may encounter `=>` in JavaScript code, which looks very similar to `<=` or `>=`. If you see `=>` it is an arrow function, which is new ES6 syntax for declaring a function expression. We will slowly introduce this syntax, especially in later courses.

- JavaScript is dynamic, and variables can change value *and* type at runtime:

```
let a; // undefined
a = 6; // 6, Number
a++; // 7, Number
a--; // 6, Number
a += 3; // 9, Number
a = 'Value=' + a; // "Value=9", String
a = !!a; // true, Boolean
a = null; // null
```

- JavaScript is a garbage collected language. Unlike C, memory automatically gets freed at runtime when variables are not longer in scope or reachable. We still need to be careful not to leak memory (i.e., hold onto data longer than necessary, or forever) and block the garbage collector from doing its job.

- Strings: JavaScript doesn't distinguish between a single `char` and a multi-character `String`--everything is a `String`. You define a `String` using either single (`'...'`), double (`"..."`) quotes. Try to pick one style and stick with it within a given file/program vs. mixing them.

- JavaScript version note: newer versions of ECMAScript also allow for the

use of template literals. Instead of `'` or `"`, a template literal uses ` (backticks), and you can also interpolate expressions.

- A JavaScript expression is any code (e.g., literals, variables, operators, and expressions) that evaluates to a single value. The value may be a `Number`, `String`, an `Object`, or a logical value.

```
let a = 10 / 2; // arithmetic expression
let b = !(10 / 2); // logical expression evaluates to false
let c = '10 ' + '/' + ' 2'; // string, evaluates to "10 / 2"
let f = function () {
  return 10 / 2;
}; // function expression, f can now be called via the () operator
let d = f(); // f() evaluates to 10/2, or the Number 5
```

- JavaScript execution flow is determined using the following four (4) basic control structures:
  - Sequential: an instruction is executed when the previous one is finished.
  - Conditional: a logical condition is used to determine which instruction will be executed next - similar to the `if` and `switch` statements in C (which JavaScript also has).
  - Looping: a series of instructions are repeatedly executed until some condition is satisfied - similar to the `for` and `while` statements in C (which JavaScript also has). There are many different types of loops in JavaScript: for example `for` loops and `while` loops, as well as ways to `break` out of loops or skip iterations with `continue`. We'll cover other types as we learn about `Object` and `Array`.
  - Transfer: jump to, or invoke a different part of the code - similar to calling a function in C.

```
/**
 * 1. Sequence example: each statement is executed one after the
other
 **/
let a = 3;
let b = 6;
let c = a + b;

/**
 * 2. Conditional examples: a decision is made based on the
evaluation of an expression,
 * and a code path (or branch) taken.
 **/
let grade;
let mark = 86;

if (mark >= 90) {
  grade = 'A+';
} else if (mark >= 80) {
  grade = 'A';
} else if (mark >= 70) {
  grade = 'B';
} else if (mark >= 60) {
  grade = 'C';
} else if (mark >= 50) {
  grade = 'D';
} else {
  grade = 'F';
}

switch (grade) {
  case 'A+':
    // do these steps if grade is A+
    break;
  case 'A':
    // do these steps if grade is A
    break;
```

Practice Exercises

Try to solve each of the following using JavaScript. If you need to `print` something, use `console.log()`, which will print the argument(s) you give it.

1. Create a variable `label` and assign it the value `"senecacollege"`. Create another variable `tld` and assign it `"ca"`. Create a third variable `domainName` that combines `label` and `tld` to produce the value `"senecacollege.ca"`.

2. Create a variable `isSeneca` and assign it a boolean value (`true` or `false`) depending on whether or not `domainName` is equal to `"senecacollege.ca"`. HINT: use `===` and don't write `true` or `false` directly.

3. Create a variable `isNotSeneca` and assign it the inverse boolean value of `isSeneca`. HINT: if `isSeneca` is `true`, `isNotSeneca` should be `false`.

4. Create four variables `byte1`, `byte2`, `byte3`, `byte4`, and assign each of these a value in the range `0-255`.

5. Convert `byte1` to a `String` using `.toString()`, and `console.log()` the result. What happens if you use `toString(2)` or `toString(16)` instead?

6. Create a variable `ipAddress` and assign it the value of combining your four `byteN` variables together, separated by `"."`. For example: `"192.168.2.1"`.

7. Create a variable `ipInt` and assign it the integer value of bit-shifting (`<<`) and adding your `byteN` variables. HINT: your `ipInt` will contain 32 bits, the first byte needs to be shifted 24 bit positions (`<< 24`) so it occupies 32-25, the second shifted 16, the third 8.

8. Create a variable `ipBinary` that contains the binary representation of the `ipInt` value. HINT: use `.toString(2)` to display the number with `1` and `0` only.

9. Create a variable `statusCode`, and assign it the value for the "I'm a teapot" HTTP status code. HINT: see https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

10. Write an `If` statement that checks to see if your `statusCode` is a `4xx` client error. HINT: use the `<`, `>`, `>=`, and/or `<=` operators to test the value

11. Write a `switch` statement that checks your `statusCode` for all possible `1xx` information responses. In each case, you should `console.log()` the response text associated with the status code, or `"unknown information response"` if the status code is not known.

12. Write a function `is2xx(status)` which takes a status code `status` (e.g., `200`) and returns `true` if the status code is a valid 2xx code.

13. Create a variable `studentName` and assign your name. Create another variable `studentAge` and assign it your age. Use `console.log()` to print out a sentence that includes both variables, like `"Alice is 20 years old."`.

14. Create a variable `isEven` and assign it a boolean value (`true` or `false`) depending on whether a given number `num` is even or not. HINT: use the modulus operator `%`.

15. Create a variable `isOdd` and assign it the inverse boolean value of `isEven`. HINT: if `isEven` is `true`, `isOdd` should be `false`.

16. Create a variable `radius` and assign it a value of `10`. Calculate the area of

a circle with this radius and assign the result to a variable `area`. HINT: use `Math.PI` and the formula `area = πr^2`.

17. Create a variable `temperatureInCelsius` and assign it a value. Convert this temperature to Fahrenheit and assign the result to a variable `temperatureInFahrenheit`. HINT: use the formula `F = C * 9/5 + 32`.

18. Create a variable `heightInFeet` and assign it a value. Convert this height to meters and assign the result to a variable `heightInMeters`. HINT: use the conversion factor `1 foot = 0.3048 meters`.

19. Create a variable `seconds` and assign it a value. Convert this time to minutes and seconds (e.g., 90 seconds becomes 1 minute and 30 seconds) and assign the result to two variables `minutes` and `remainingSeconds`.

20. Create a variable `score` and assign it a value. Write an `if` statement that checks if the score is an A (90-100), B (80-89), C (70-79), D (60-69), or F (below 60) and assigns the result to a variable `grade`.

21. Write a `switch` statement that checks the value of a variable `day` and `console.log()`s whether it is a weekday or weekend. HINT: `day` can be a value from 1 (Monday) to 7 (Sunday).

22. Write a function `isPositive(num)` which takes a number `num` and returns `true` if the number is positive and `false` otherwise.

23. Write a function `isLeapYear(year)` which takes a year `year` and returns `true` if the year is a leap year and `false` otherwise. HINT: a leap year is divisible by 4, but not by 100, unless it is also divisible by 400.

24. Write a function `getDayOfWeek(day)` which takes a number `day` (from 1 to 7) and returns the day of the week as a string (e.g., "Monday").

25. Write a function `getFullName(firstName, lastName)` which takes two strings `firstName` and `lastName` and returns the full name as a single string.

26. Write a function `getCircleArea(radius)` which takes a number `radius` and returns the area of a circle with that radius.

27. Write a function `getHypotenuse(a, b)` which takes two numbers `a` and `b` (the lengths of the two sides of a right triangle) and returns the length of the hypotenuse. HINT: use the Pythagorean theorem and `Math.sqrt()` to calculate the square root.

After you try writing these yourself, take a look at a possible solution.

Functions

A function is a *subprogram*, or a smaller portion of code that can be called (i.e., invoked) by another part of your program, another function, or by the environment in response to some user or device action (e.g., clicking a button, a network request, the page closing). Functions *can* take values (i.e., arguments) and may *return* a value.

Functions are first-class members of JavaScript, and play a critical role in developing JavaScript programs. JavaScript functions can take other functions as arguments, can return functions as values, can be bound to variables or `Object` properties, and can even have their own properties. We'll talk about more of this when we visit JavaScript's object-oriented features.

Learning to write code in terms of functions takes practice. JavaScript supports functional programming. Web applications are composed of lots of small components that need to get wired together using functions, have to share data (i.e., state), and interoperate with other code built into the browser, or in third-party frameworks, libraries, and components.

We use JavaScript functions in a number of ways. First, we encapsulate a series of statements into higher-order logic, giving a name to a set of repeatable steps we can call in different ways and places in our code. Second, we use them to define actions to be performed in response to events, whether user initiated or triggered by the browser. Third, we use them to define behaviours for objects, what is normally called a *member function* or *method*. Fourth, we use them to define *constructor* functions, which are used to create new objects. We'll look at all of these in the coming weeks.

Before we dive into that, we'll try to teach you that writing many smaller functions is often better than having a few large ones. Smaller code is easier to

test, easier to understand, and generally has fewer bugs.

User-defined Functions

JavaScript has many built-in functions, which we'll get to below; however, it also allows you to write your own and/or use functions written by other developers (libraries, frameworks).

These user-defined functions can take a number of forms.

Function Declarations

The first is the *function declaration*, which looks like this:

```
// The most basic function, a so-called NO OPERATION function
function noop() {}

// square function accepts one parameter `n`, returns its value
squared.
function square(n) {
  return n * n;
}

// add function accepts two parameters, `a` and `b`, returns their
sum.
function add(a, b) {
  return a + b;
}
```

Here the `function` keyword initiates a *function declaration*, followed by a *name*, a *parameter list* in round parenthesis, and the function's *body* surrounded by curly braces. There is no semi-colon after the function body.

Function Expressions

The second way to create a function is using a *function expression*. Recall that expressions evaluate to a value: a function expression evaluates to a `function` Object. The resulting value is often bound (i.e., assigned) to a variable, or used as a parameter.

```
let noop = function () {};

let square = function (n) {
  return n * n;
};

let add = function add(a, b) {
  return a + b;
};
```

A few things to note:

- The function's *name* is often omitted. Instead we return an *anonymous function* and bind it to a variable. We'll access it again via the variable name. In the case of recursive functions, we sometimes include it to make it easier for functions to call themselves. You'll see it done both ways.
- We *did* use a semi-colon at the end of our function expression. We do this to signify the end of our assignment statement `let add = ... ;`.
- In general, *function declarations* are likely a better choice (when you can choose) due to subtle errors introduced with declaration order and hosting (see below); however, both are used widely and are useful.

Arrow Functions

Modern JavaScript also introduces a new function syntax called an Arrow Function or "Fat Arrow". These functions are more terse, using the `=>` notation (not to be confused with the `<=` and `>=` comparison operators):

```
let noop = () => {};

let square = (n) => n * n;

let add = (a, b) => a + b;
```

When you see `let add = (a, b) => a + b;` it is short-hand for `let add = function(a, b) { return a + b; }`, where `=>` replaces the `function` keyword and comes *after* the parameter list, and the `return` keyword is optional, when functions return a single value.

Arrow functions also introduce some new semantics for the `this` keyword, which we'll address later.

You should be aware of Arrow functions, since many web developers use them heavily. However, don't feel pressure to use them yet if you find their syntax confusing.

Parameters and `arguments`

Function definitions in both cases take parameter lists, which can be empty, single, or multiple in length. Just as with variable declaration, no type information is given:

```
function emptyParamList() {}

function singleParam(oneParameter) {}

function multipleParams(one, two, three, four) {}
```

A function can *accept* any number of arguments when it is called, including none. This would break in many other languages, but not JavaScript:

```
function log(a) {
  console.log(a);
}

log('correct'); // logs "correct"
log('also', 'correct'); // logs "also"
log(); // logs undefined
```

Because we can invoke a function with any number of arguments, we have to write our functions carefully, and test things before we make assumptions. How can we deal with a caller sending 2 vs. 10 values to our function?

One way we do this is using the built-in `arguments` Object.

Every function has an implicit `arguments` variable available to it, which is an array-like object containing all the arguments passed to the function. We can use `arguments.length` to obtain the actual number of arguments passed to the function at runtime, and use array index notation (e.g., `arguments[0]`) to access an argument:

```
function log(a) {
  console.log(arguments.length, a, arguments[0]);
}
```

We can use a loop to access all arguments, no matter the number passed:

```
function sum() {
  const count = arguments.length;
  let total = 0;
  for (let i = 0; i < count; i++) {
    total += arguments[i];
  }
  return total;
}

sum(1); // 1
sum(1, 2); // 3
sum(1, 2, 3, 4); // 10
```

You may have wondered previously how `console.log()` can work with one, two, three, or more arguments. The answer is that all JavaScript functions work this way, and you can use it to "overload" your functions with different argument patterns, making them useful in more than one scenario.

Parameters and `...`

Modern JavaScript also supports naming the "rest" of the parameters passed to a function. These Rest Parameters allow us to specify that all final arguments to a function, no matter how many, should be available to the function as a named `Array`.

There are some advantages to *not* using the implicit `arguments` keyword, which rest parameters provide.

We can convert the example above to this, naming our arbitrary list of "numbers":

```
function sum(...numbers) {
  let total = 0;
  for (let i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }
  return total;
}
```

Dealing with Optional and Missing Arguments

Because we *can* change the number of arguments we pass to a function at runtime, we also have to deal with missing data, or optional parameters. Consider the case of a function to calculate a player's score in a video game. In some cases we may want to double a value, for example, as a bonus for doing some action a third time in a row:

```
function updateScore(currentScore, value, bonus) {
  return bonus ? currentScore + value * bonus : currentScore +
value;
}

updateScore(10, 3);
updateScore(10, 3);
updateScore(10, 3, 2);
```

Here we call `updateScore` three different times, sometimes with 2 arguments, and once with 3. Our `updateScore` function has been written so it will work in both cases. We've used a conditional ternary operator to decide whether or not to add an extra bonus score. When we say `bonus ? ... : ...` we are checking to see if the `bonus` argument is *truthy* or *falsy*--that is, did the caller provide a value for it? If they did, we do one thing, if not, we do another.

Here's another common way you'll see code like this written, using a default value:

```
function updateScore(currentScore, value, bonus) {
  // See if `bonus` is truthy (has a value or is undefined) and
use it, or default to 1
  bonus = bonus || 1;
  return currentScore + value * bonus;
}
```

In this case, before we use the value of `bonus`, we do an extra check to see if it actually has a value or not. If it does, we use that value as is; but if it doesn't, we instead assign it a value of `1`. Then, our calculation will always work, since multiplying the value by `1` will be the same as not using a bonus.

The idiom `bonus = bonus || 1` is very common in JavaScript. It uses the Logical Or Operator `||` to test whether `bonus` evaluates to a value or not, and prefers that value if possible to the fallback default of `1`. We could also have written it out using an `if` statements like these:

```
function updateScore(currentScore, value, bonus) {
  if (bonus) {
    return currentScore + value * bonus;
  }
  return currentScore + value;
}

function updateScore(currentScore, value, bonus) {
  if (!bonus) {
    bonus = 1;
  }
  return currentScore + value * bonus;
}
```

JavaScript programmers tend to use the `bonus = bonus || 1` pattern because it is less repetitive, using less code, and therefore less likely to introduce bugs. We could shorten it even further to this:

```
function updateScore(currentScore, value, bonus) {
  return currentScore + value * (bonus || 1);
}
```

Because this pattern is so common, modern JavaScript has added a built-in way to handle Default Parameters. Instead of using `||` notation in the body of the function, we can specify a default value for any named parameter when it is declared. This frees us from having to check for, and set default values in the function body. Using default parameters, we could convert our code above to this:

```
function updateScore(currentScore, value, bonus = 1) {
  return currentScore + value * bonus;
}
```

Now, if `bonus` has a value (i.e., is passed as a parameter), we use it; otherwise, we use `1` as a default.

Return Value

Functions always *return* a value, whether implicitly or explicitly. If the `return` keyword is used, the expression following it is returned from the function. If it is omitted, the function will return `undefined`:

```
function implicitReturnUndefined() {
  // no return keyword, the function will return `undefined`
```

Function Naming

Functions are typically named using the same rules we learned for naming any variable: `camelCase` and using the set of valid letters, numbers, etc. and avoiding language keywords.

Function declarations always give a name to the function, while function expressions often omit it, using a variable name instead:

```
// Name goes after the `function` keyword in a declaration
function validateUser() {
    ...
}

// Name is used only at the level of the bound variable, function
is anonymous
let validateUser = function() {
    ...
};

// Name is repeated, which is correct but not common. Used with
recursive functions
let validateUser = function validateUser() {
    ...
};

// Names are different, which is also correct, but not common as
it can lead to confusion
let validateUser = function validate() {
    // the validate name is only accessible here, within the
function body
    ...
};
```

Because JavaScript allows us to bind function objects (i.e., result of function expressions) to variables, it is common to create functions without names, but immediately pass them to functions as arguments. The only way to use this function is via the argument name:

```
// The parameter `fn` will be a function, and `n` a number
function execute(fn, n) {
  // Call the function referred to by the argument (i.e, variable)
`fn`, passing `n` as its argument
  return fn(n);
}

// 1. Call the `execute` function, passing an anonymous function,
which squares its argument, and the value 3
execute(function (n) {
  return n * n;
}, 3);

// 2. Same thing as above, but with different formatting
execute(function (n) {
  return n * n;
}, 3);

// 3. Same thing as above, using an Arrow Function
execute((n) => n * n, 3);

let doubleIt = function (num) {
  return num * 2;
};

// 4. Again call `execute`, but this time pass `doubleIt` as the
function argument
execute(doubleIt, 3);
```

We can also use functions declared via function declarations used this way,

and bind them to variables:

```
function greeting(greeting, name) {
  return greeting + ' ' + name;
}

var sayHi = greeting; // also bind a reference to greeting to
sayHi

// We can now call `greeting` either with `greeting()` or
`sayHi()`
console.log(greeting('Hello', 'Steven'));
console.log(sayHi('Hi', 'Kim'));
```

JavaScript treats functions like other languages treat numbers or booleans, and lets you use them as values. This is a very powerful feature, but can cause some confusion as you get started with JavaScript.

You might ask why we would ever choose to define functions using variables. One common reason is to swap function implementations at runtime, depending on the state of the program. Consider the following code for displaying the user interface depending on whether the user is logged in or not:

```
// Display partial UI for guests and non-authenticated users,
hiding some features
function showUnauthenticatedUI() {
    ...
}

// Display full UI for authenticated users
function showAuthenticatedUI() {
    ...
```

Invoking Functions, the Execution Operator

In many of the examples above, we've been invoking (i.e., calling, running, executing) functions but haven't said much about it. We invoke a function by using the `()` operator:

```
let f = function () {
  console.log('f was invoked');
};
f();
```

In the code above, `f` is a variable that is assigned the value returned by a function expression. This means `f` is a regular variable, and we can use it like any other variable. For example, we could create another variable and share its value:

```
let f = function () {
  console.log('f was invoked');
};
let f2 = f;
f(); // invokes the function
f2(); // also invokes the function
```

Both `f` and `f2` refer to the the same function object. What is the difference between saying `f` vs. `f()` in the line `let f2 = f;`? When we write `f()` we are really saying, "Get the value of `f` (the function referred to) and invoke it." However, when we write `f` (without `()`), we are saying, "Get the value of `f` (the function referred to)" so that we can do something with it (assign it to another variable, pass it to a function, etc).

The same thing is true of function declarations, which also produce `function`

Objects:

```
function f() {
  console.log('f was invoked');
}
let f2 = f;
f2(); // also invokes the function
```

The distinction between referring to a function object via its bound variable name (`f`) vs invoking that same function (`f()`) is important, because JavaScript programs treat functions as *data*, just as you would a `Number`. Consider the following:

```
function checkUserName(userName, customValidationFn) {
  // If `customValidationFn` exists, and is a function, use that
to validate `userName`
  if (customValidationFn && typeof customValidationFn ===
'function') {
    return customValidationFn(userName);
  }
  // Otherwise, use a default validation function
  return defaultValidationFn(userName);
}
```

Here the `checkUserName` function takes two arguments: the first a `String` for a username; the second an optional (i.e., may not exist) function to use when validating this username. Depending on whether or not we are passed a function for `customValidationFn`, we will either use it, or use a default validation function (defined somewhere else).

Notice the line `if(customValidationFn && typeof customValidationFn === 'function') {` where `customValidationFn` is used like any other variable

(accessing the value it refers to vs. doing an invocation), to check if it has a value, and if its value is actually a function. Only then is it save to invoke it.

It's important to remember that JavaScript functions aren't executed until they are called via the invocation operator, and may also be used as values without being called.

Built-in/Global Functions

JavaScript provides a small number of built-in global functions for working with its data types, for example:

- `parseInt()`
- `parseFloat()`
- `isNaN()`
- `isFinite()`
- `decodeURI()`
- `decodeURIComponent()`
- `encodeURI()`
- `encodeURIComponent()`

There are also global functions that exist for historical reasons, but should be avoided for performance, usability, and/or security reasons:

- `eval()` dangerous to parse and run user-defined strings
- `prompt()` and `alert()` synchronous calls that block the UI thread.

Most of JavaScripts "standard library" comes in the form of *methods* on global objects vs. global functions. A *method* is a function that is bound to a variable belonging to an object, also known as a *property*. We'll be covering these in

more depth later, but here are some examples

- `console.*`. There are quite a few worth learning, but here are some to get you started: _ `console.log()`, `console.warn()`, and `console.error()` _ `console.assert()` _ `console.count()` _ `console.dir()`
- `Math.*`
  - `Math.abs()`
  - `Math.max()`
  - `Math.min()`
  - `Math.random()`
  - `Math.round()`
- `Date.*`
  - `Date.now()`
  - `Date.getTime()`
  - `Date.getMonth()`
  - `Date.getDay()`
- `JSON.*`
  - `JSON.parse()`
  - `JSON.stringify()`

Much of web programming is done using `Objects` and calling their methods. JavaScript is a small language, but the ecosystem of `Objects`, APIs, libraries, and frameworks allows it to do anything.

Suggested Readings

- ExploringJS, Chapter 15. Functions and Chapter 16. Variables: Scopes, Environments, and Closures
- Eloquent JavaScript, Chapter 3. Functions

- Functions Guide and Reference on MDN.

Scope

JavaScript variables were historically *declared* with the `var` keyword. Modern JavaScript has switched to `let`, `const`. The way each works is different, and it's important to understand these differences.

We often *assign* a value when we *declare* it, though we don't have to do both at once:

```
let x; // declared, no assignment (value is `undefined`)
x = 7; // assignment of previously declared variable
let y = x; // declaration and assignment combined
```

A variable always has a *scope*, which is the location(s) in the code where it is usable. Consider the variables `total` and `value`, as well as the `add` function below:

```
var total = 7; // global variable, accessible everywhere

function add(n) {
  var value = total + n; // local variable, accessible anywhere
within the function only
  return value;
}

console.log('Total is', total); // Works, because `total` is in
the same scope
console.log('Value is', value); // `undefined`, since `value`
isn't defined in this scope
console.log('New Total', add(16)); // Works, because `add` is
defined in the same scope
```

When using the `var` keyword, variables use *function scope*, while variables declared with `let` and `const` use *block scope*. Coming from C/C++, using `let` and `const` will likely feel more familiar`:

```
int main()
{
  {
      int x = 10;        // x is declared with block scope
  }
  {
      printf("%d", x);  // Error: x is not accessible here
  }
  return 0;
}
```

Now in JavaScript:

```
function main() {
  {
    var x = 10; // x is declared in a block, but is scoped to
`main`
  }
  {
    console.log(x); // works, because `x` is accessible everywhere
in `main`
  }
}
```

Because variables declared using `var` have **function scope**, programmers tended to define them at the top of their functions. They don't strictly need to do this, since JavaScript will *hoist* or raise all variables declared with `var` in a function to the top of the function's scope:

```
function f() {
  var y = x + 1;
  var x = 2;
}
```

At runtime, this will be transformed into the following:

```
function f() {
    var x;          // declaration is hoisted (but not assignment)
to the top

    var y = x + 1;  // `NaN`, since `undefined` + 1 can't be
resolved
    x = 2;          // note: `x` is not declared above, only the
assignment is now here
```

This also happens when we forget to declare a local variable:

```
function f() {
  x = 2; // `x` is assigned a value, but not declared
  return x + 1;
}
```

At runtime, this will be transformed into the following:

```
var x; // `x` is not found in the scope of `f`, so it becomes
global

function f() {
  x = 2;
  return x + 1;
}
```

The previous example introduces another important concept with JavaScript scopes, namely, that scopes can be *nested* within one another. Hoisting is moving variable declarations to the beginning of a scope. For example, function declarations are hoisted completely, which means we can call a function *before* we declare it.

```
f(); // this will work, as f's declaration gets hoisted
function f() {}
f(); // this will also work, because f has been declared as you
expect.

g(); // this will not work, since g's declaration will be hoisted,
but not the assignment.
var g = function () {};
```

Many of the confusing aspects of function scope and hoisting are solved by using `let` and `const`, which work at the block level instead. Consider these two loops:

```
// Version 1 using var
for (var i = 0; i < 10; i++) {
  console.log('The value of i is ' + i);
}

// Version 2 using let
for (let i = 0; i < 10; i++) {
  console.log('The value of i is ' + i);
}
```

In the preceding code, the scope of `i` is different in version 1 vs. 2. In version 1, the declaration of `i` will actually cause a variable to be created in the scope of the owning function. This may or may not be what you expect (i.e., the variable `i` will exist outside the loop). In version 2, this is not the case, and `i`

is scoped to the function body only (i.e., you can't access it before or after the loop).

We're discussing both function and block scopes because JavaScript supports each of them, and code you'll work on will use both methods. It's important to understand each approach.

For new code that you write, you are encouraged to prefer `let` and `const` and use block scope.

Overwriting Variables in Child Scopes

Since variables defined with `var` have function scope, and because functions can be nested, we have to be careful when naming our variables and arguments so as to not overwrite a variable in a parent scope. Or, we can use this to temporarily do exactly that. In both cases, we need to understand how nested scopes work.

Consider the the following code, where a variable named `x` is used in three different scopes. What will be printed to the `console` when `child` is called?

```
var x = 1;

function parent() {
  var x = 2;

  function child(x) {
    console.log(x);
  }
```

The first declaration of `x` creates a global variable (i.e., available in every scope). Then, in `parent` we re-declare `x`, creating a new local variable, which overwrites (or hides) the global variable `x` in this scope (i.e., within the body of `parent`). Next, we define yet another scope for `child`, which also uses `x` as the name of its only argument (essentially another local variable). When we do `child(3)`, we are binding the value `3` to the `x` argument defined for the scope of `child`, and in so doing yet again overwriting the parent `x`. In the end, the console will show `3`.

We can do this in error as well, and cause unexpected behaviour:

```
var total = 5;

function increase(n) {
  var total = n + n;
}

increase(50);
console.log(total);
```

Here we expect to see `100` but instead will get `5` on the `console.` The problem is that we have redefined, and thus overwritten `total` inside the `increase` function. During the call to `increase`, the new local variable `total` will be used, and then go out of scope. After the function completes, the original global variable `total` will again be used.

Closures

A closure is a function that has *closed over* a scope, retaining this scope even after it would otherwise disappear through the normal rules of execution. In the following function, the variable `x` goes out of scope as soon as the function

finishes executing:

```
function f() {
  var x = 7;
  return x * 2;
  // After this return, and f completes, `x` will no longer be
available.
}
```

In JavaScript, functions have access not only to their own local variables, but also to any variables in their parents' scope. That is, if a variable is used (referenced) but not declared in a function, JavaScript will visit the parent scope to find the variable. This can happen for any number of child/parent levels up to the global level.

The following is an example of this, and probably one you've seen before:

```
var x = 7;

function f() {
  return x * 2; // `x` not declared here, JS will look in the
parent scope (global)
}
```

Consider this example:

```
function parent() {
  var x = 7;

  function child() {
    return x * 2;
  }
```

Here `x` is used in `child`, but declared in `parent`. The `child` function has access to all variables in its own scope, plus those in the `parent` scope. This nesting of scopes relies on JavaScript's function scope rules, and allows us to share data.

Sometimes we need to capture data in a parent scope, and retain it for a longer period of time than would otherwise be granted for a given invocation. Consider this example:

```
function createAccumulator(value) {
  return function (n) {
    value += n;
    return value;
  };
}

var add = createAccumulator(10);
add(1); // returns 11
add(2); // returns 13
```

Here the `createAccumulator` function takes an argument `value`, the initial value to use for an accumulator function. It returns an anonymous function which takes a value `n` (a `Number`) and adds it to the `value` before returning it. The `add` function is created by invoking `createAccumulator` with the initial `value` of `10`. The function that is returned by `createAccumulator` has access to `value` in its parent's scope. Normally, `value` would be destroyed as soon as `createAccumulator` finished executing. However, we have created a *closure* to capture the variable `value` in a scope that is now attached to the function we're creating and returning. As long as the returned function exists (i.e., as long as `add` holds on to it), the variable `value` will continue to exist in our child function's scope: the variables that existed when this function was created continue to live on like a memory, attached to the lifetime of the

returned function.

Closures make it possible to *associate* some *data* (i.e., the environment) with a function that can then operate on that data. We see similar strategies in pure object-oriented languages, where data (properties) can be associated with an object, and functions (methods) can then operate on that data. Closures play a somewhat similar role, however, they are more lightweight and allow for dynamic (i.e., runtime) associations.

By connecting data and functionality, closures help to reduce global variables, provide ways to "hide" data, allow a mechanism for creating private "methods", avoid overwriting other variables in unexpected ways.

As we go further with JavaScript and web programming, we will encounter many instances where closures can be used to manage variable lifetimes, and associated functions with specific objects. For now, be aware of their existence, and know that it is an advanced concept that will take some time to fully master. This is only our first exposure to it.

Another way we'll see closures used, is in conjunction with Immediately-Invoked Function Expressions (IIFE). Consider the following rewrite of the code above:

```
let add = (function (value) {
  return function (n) {
    value += n;
    return value;
  };
})(10);

add(1); // returns 11
add(2); // returns 13
```

Here we've declared `add` to be the value of invoking the anonymous function expression written between the first `(...)` parentheses. In essence, we have created a function that gets executed immediately, and which returns another function that we will use going forward in our program.

This is an advanced technique to be aware of at this point, but not one you need to master right away. We'll see it used, and use it ourselves, in later weeks to to avoid global variables, simulate block scope in JavaScript, and to choose or generate function implementations at runtime (e.g., polyfill).

Practice Exercises

For each of the following, write a function that takes the given arguments, and returns or produces (e.g., `console.log`) the given result.

1. Given `r` (radius) of a circle, calculate the area of a circle (A = π _ r _ r).

2. Simulate rolling a dice using `random()`. The function should allow the caller to specify any number of sides, but default to 6 if no side count is given: `roll()` (assume 6 sided, return random number between 1 and 6) vs. `roll(50)` (50 sided, return number between 1 and 50).

3. Write a function that converts values in Celcius to Farenheit: `convert(0)` should return `"32 F"`.

4. Modify your solution to the previous function to allow a second argument: `"F"` or `"C"`, and use that to determine what the scale of the value is, converting to the opposite: `convert(122, "F")` should return `"50 C"`.

5. Function taking any number of arguments (`Number`s), returning `true` if they are all less than 50: `isUnder50(1, 2, 3, 5, 4, 65)` should return `false`.

6. Function allowing any number of arguments (`Number`s), returning their sum: `sum(1, 2, 3)` should return `6`.

7. Function allowing any number of arguments of any type, returns `true` only if none of the arguments is falsy. `allExist(true, true, 1)` should return `true`, but `allExist(1, "1", 0)` should return `false`.

8. Function to create a JavaScript library name generator:

`generateName("dog")` should return `"dog.js"`

9. Function to check if a number is a multiple of 3 (returns `true` or `false`)
10. Check if a number is between two other numbers, being inclusive if the final argument is true: `checkBetween(66, 1, 50, true)` should return `false`.
11. Function to calculate the HST (13%) on a purchase amount
12. Function to subtract a discount % from a total. If no % is given, return the original value.
13. Function that takes a number of seconds as a `Number`, returning a `String` formatted like `"X Days, Y Hours, Z Minutes"` rounded to the nearest minute.
14. Modify your solution above to only include units that make sense: `"1 Minute"` vs. `"3 Hours, 5 Minutes"` vs. `"1 Day, 1 Hour, 56 Minutes"` etc
15. Function that takes any number of arguments (`Number`s), and returns them in reverse order, concatenated together as a String: `flip(1, 2, 3)` should return `"321"`
16. Function that takes two `Number`s and returns their sum as an `Integer` value (i.e., no decimal portion): `intSum(1.6, 3.333333)` should return `4`
17. Function that returns the number of matches found for the first argument in the remaining arguments: `findMatches(66, 1, 345, 2334, 66, 67, 66)` should return `2`
18. Function to log all arguments larger than `255`: `showOutsideByteRange(1, 5, 233, 255, 256, 0)` should log `256` to the `console`

19. Function that takes a `String` and returns its value properly encoded for use in a URL. `prepareString("hello world")` should return `"hello%20world"`

20. Using the previous function, write an enclosing function that takes any number of `String` arguments and returns them in encoded form, concatenated together like so: `"?...&...&..."` where "..." are the encoded strings. `buildQueryString("hello world", "goodnight moon")` should return `"?hello%20world&goodnight%20moon"`

21. Function that takes a `Function` followed by any number of `Number`s, and applies the function to all the numbers, returning the total: `applyFn(function(x) { return x * x;}, 1, 2, 3)` should return 14.

After you try writing these yourself, take a look at a possible solution

Introduction

In languages like C, we are used to thinking about data types separately from the functions that operate upon them. We declare variables to hold data in memory, and call functions passing them variables as arguments to operate on their values.

In object-oriented languages like JavaScript, we are able to combine data and functionality into higher order types, which both contain data and allow us to work with that data. In other words, we can pass data around in a program, and all the functionality that works on that data travels with it.

Let's consider this idea by looking at strings in C vs. JavaScript. In C a string is a null terminated (`\0`) array of `char` elements, for example:

```
const char name1[31] = "My name is Arnold";
const char name2[31] = {'M','y',' ','n','a','m','e',' ','i','s',' ','A','r','n','o','l','d','\0'};
```

With C-style strings, we perform operations using standard library functions, for example `string.h`:

```
#include <string.h>

int main(void)
{
    char str[31];        // declare a string
    ...
    strlen(str);         // find the length of a string str
    strcpy(str2, str);   // copy a string
    strcmp(str2, str);   // compare two strings
```

JavaScript also allows us to work with strings, but because JavaScript is an object-oriented language, a JavaScript `String` is an `Object` with various properties and methods we can use for working with text.

One way to think about `Object`s like `String` is to imagine combining a C-string's data type with the functions that operate on that data. Instead of needing to specify *which* string we want to work with, all functions would operate a particular *instance* of a string. Another way to look at this would be to imagine that the data and the functions for working with that data are combined into one more powerful type. If we could do this in C, we would be able to write code that looked more like this:

```
String str = "Hello"; // declare a string

int len = str.len;    // get the length of str
str.cmp(str2);        // compare str and str2
str = str.cat("..."); // concatenate "..." onto str
```

In the made-up code above, the *data* (`str`) is attached to functionality that we can call via the `.*` notation. Using `str.*`, we no longer need to indicate to the functions which string to work with: all string functions work on the string data to which they are attached.

This is very much how `String` and other `Object` types work in JavaScript. By combining the string character data and functionality into one type (i.e., a `String`), we can easily create and work with text in our programs.

Also, because we work with strings at a higher level of abstraction (i.e., not as arrays of `char`), JavaScript deals with memory management for us, allowing our strings to grow or shrink at runtime.

Strings

Here are a few examples of how you can declare a `String` in JavaScript, first using a string literal, followed by a call to the `new` operator and the `String` object's constructor function:

```
/*
 * JavaScript String Literals
 */
let s = 'some text'; // single-quotes
let s1 = 'some text'; // double-quotes
let s2 = `some text`; // template literal using back-ticks
let unicode =
  '¿¿ español Deutsch English ¿¿¿¿¿¿¿¿ العربية português
¿¿¿¿¿ русский ¿¿¿ ¿¿¿¿¿¿ ¿¿¿ ¿¿¿¿¿ עברית'; // non-ASCII
characters

/*
 * JavaScript String Constructor: `new String()` creates a new
instance of a String
 */
let s3 = new String('Some Text');
let s4 = new String('Some Text');
```

If we want to convert other types to a `String`, we have a few options:

```
let x = 17;
let s = '' + x; // concatenate with a string (the empty string)
let s2 = String(x); // convert to String. Note: the `new` operator
is not used here
let s3 = x.toString(); // use a type's .toString() method
```

Whether you use a literal or the constructor function, in all cases you will be able to use the various functionality of the `String` type.

String Properties and Methods

- `s.length` - will tell us the length of the string (UTF-16 code units)
- `s.charAt(1)` - returns the character at the given position (UTF-16 code unit). We can also use `s[1]` and use an index notation to get a particular character from the string.
- `s.concat()` - returns a new string created by concatenating the original with the given arguments.
- `s.padStart(2, '0)` - returns a new string padded with the given substring until the length meets the minimum length given. See also `s.padEnd()`.
- `s.includes("tex")` - returns `true` if the search string is found within the string, otherwise `false` if not found.
- `s.startsWith("some")` - returns `true` if the string starts with the given substring, otherwise `false`.
- `s.endsWith("text")` - returns `true` if the string ends with the given substring, otherwise `false`.
- `s.indexOf("t")` - returns the first index position of the given substring within `s`, or `-1` if the substring is not found within `s`. See also `s.lastIndexOf()`
- `s.match(regex)` - tries to match a regular expression against the string, returning the matches. See discussion of RegExp below.
- `s.replace(regex, "replacement")` - returns a new string with the first occurrence of a matched RegExp replaced by the replacement text. See also `s.replaceAll()`, which replaces *all* occurrences.
- `s.slice(2, 3)` - returns a new string extracted (sliced) from within the

original string. A beginning index and (optional) end index mark the position of the slice.

- `s.split()` - returns an Array (see discussion below) of substrings by splitting the original string based on the given separator (`String` or `RegExp`).
- `s.toLowerCase()` - returns a new string with all characters converted to lower case.
- `s.toUpperCase()` - returns a new string with all characters converted to upper case.
- `s.trim()` - returns a new string with leading and trailing whitespace removed.

> JavaScript Version Note: modern JavaScript also supports template literals, also sometimes called template *strings*. Template literals use back-ticks instead of single- or double-quotes, and allow you to interpolate JavaScript expressions. For example:

```
let a = 1;
let s = 'The value is ' + 1 * 6;
// Use ${...} to interpolate the value of an expression into a string
let templateVersion = `The value is ${1 * 6}`;
```

Arrays

An `Array` is an `Object` with various properties and methods we can use for working with lists in JavaScript.

Declaring JavaScript Arrays

Like creating a `String`, we can create an `Array` in JavaScript using either a literal or the `Array` constructor function:

```
let arr = new Array(1, 2, 3); // array constructor
let arr2 = [1, 2, 3]; // array literal
```

Like arrays in C, a JavaScript `Array` has a length, and items contained within it can be accessed via an index:

```
let arr = [1, 2, 3];
let len = arr.length; // len is 3
let item0 = arr[0]; // item0 is 1
```

Unlike languages such as C, a JavaScript `Array` can contain any type of data, including mixed types:

```
let list = [0, '1', 'two', true];
```

JavaScript `Array`s can also contain holes (i.e., be missing certain elements), change size dynamically at runtime, and we don't need to specify an initial size:

```
let arr = []; // empty array
arr[5] = 56; // element 5 now contains 56, and arr's length is now
6
```

> NOTE: a JavaScript `Array` is really a **map**, which is a data structure that associates values with unique keys (often called a key-value pair). JavaScript arrays are a special kind of map that uses numbers for the keys, which makes them look and behave very much like arrays in other languages. We will encounter this **map** structure again when we look at how to create `Object`s.

Accessing Elements in an Array

Like arrays in C, we can use index notation to obtain an element at a given index:

```
let numbers = [50, 12, 135];
let firstNumber = numbers[0];
let lastNumber = numbers[numbers.length - 1];
```

JavaScript also allows us to use a technique called Destructuring Assignment to unpack values in an Array (or Object, see below) into distinct variables. Consider each of the following methods, both of which accomplish the same goal:

```
// Co-ordinates for Seneca's Newnham Campus
let position = [43.796, -79.3486];

// Separate the two values into their own unique variables.
```

This technique is useful when working with structured data, where you know exactly how many elements are in an array, and need to access them:

```
let dateString = `17/02/2001`;
let [day, month, year] = dateString.split('/');
console.log(`The day is ${day}, month is ${month}, and year is
${year}`);
```

Here we `.split()` the string `'17/02/2001'` at the `'/'` character, which will produce the Array `['17', '02', '2001']`. Next, we destructure this Array's values into the variables `day`, `month`, `year`.

You can also ignore values (i.e., only unpack the one or ones you want):

```
let dateString = `17/02/2001`;
// Ignore the first index in the array, unpack only position 1 and
2
let [, month, year] = dateString.split('/');
console.log(`The month is ${month}, and year is ${year}`);

let emailAddress = `jsmith@myseneca.ca`;
// Only unpack the first position, ignoring the second
let [username] = emailAddress.split('@');
console.log(`The username for ${emailAddress} is ${username}`);
```

`Array` Properties and Methods

- `arr.length` - a property that tells us the number of elements in the array.

Methods that modify the original array

- `arr.push(element)` - a method to add one (or more) element(s) to the end of the array. Using `push()` modifies the array (increasing its size). You can also use `arr.unshift(element)` to add one (or more) element to the *start* of the array.
- `arr.pop()` - a method to remove the last element in the array and return it. Using `pop()` modifies the array (reducing its size). You can also use `arr.shift()` to remove the *first* element in the array and return it.

Methods that do not modify the original array

- `arr.concat([4, 5], 6)` - returns a new array with the original array joined together with other arrays or values provided.
- `arr.includes(element)` - returns `true` if the array includes the given element, otherwise `false`.
- `arr.indexOf(element)` - returns the index of the given element in the array, if it exists, otherwise `-1` (meaning not found).
- `arr.join("\n")` - returns a string created by joining (concatenating) all elements in the array with the given delimiter (`String`).

Methods for iterating across the elements in an Array

JavaScript's `Array` type also provides a long list of useful methods for working with list data. All of these methods work in roughly the same way:

```
// Define an Array
```

JavaScript will call the given function on every element in the array, one after the other. Using these methods, we are able to work with the elements in an Array instead of only being able to do things with the Array itself.

As a simple example, let's copy our `list` Array and add 3 to every element. We'll do it once with a for-loop, and the second time with the `forEach()` method:

```
// Create a new Array that adds 3 to every item in list, using a
for-loop
let listCopy = [];

for (let i = 0; i < list.length; i++) {
  let element = list[i];
  element += 3;
  listCopy.push(element);
}
```

Now the same code using the `Array`'s `forEach()` method:

```
let listCopy = [];

list.forEach(function (element) {
  listCopy.push(element + 3);
});
```

We've been able to get rid of all the indexing code, and with it, the chance for off-by-one errors. We also don't have to write code to get the element out of the list: we just use the variable passed to our function.

These `Array` methods are so powerful that there are often functions that do exactly what we need. For example, we could shorten our code above even further but using the `map()` method. The `map()` method takes one `Array`, and

calls a function on every element, creating and returning a new `Array` with those elements:

```
let listCopy = list.map(function (element) {
  return element + 3;
});
```

Here are some of the `Array` methods you should work on learning:

- `arr.forEach()` - calls the provided function on each element in the array.
- `arr.map()` - creates and returns a new array constructed by calling the provided function on each element of the original array.
- `arr.find()` - finds and returns an element from the array which matches a condition you define. See also `arr.findLast()`, `arr.findIndex()`, and `arr.findLastIndex()`, which all work in similar ways.
- `arr.filter()` - creates and returns a new array containing only those elements that match a condition you define in your function.
- `arr.every()` - returns `true` if all of the elements in the array meet a condition you define in your function.

There are more `Array` methods you can learn as you progress with JavaScript, but these will get you started.

Iterating over String, Array, and other collections

The most familiar way to iterate over a `String` or `Array` works as you'd expect:

```
let s = 'Hello World!';
for (let i = 0; i < s.length; i++) {
  let char = s.charAt(i);
  console.log(i, char);
  // Prints:
  // 0, H
  // 1, e
  // 2, l
  // ...
}

let arr = [10, 20, 30, 40];
for (let i = 0; i < arr.length; i++) {
  let elem = arr[i];
  console.log(i, elem);
  // Prints:
  // 0, 10
  // 1, 20
  // 2, 30
  // ...
}
```

The standard `for` loop works, but is not the best we can do. Using a `for` loop is prone to various types of errors: off-by-one errors, for example. It also requires extra code to convert an index counter into an element.

An alternative approach is available in ES6, `for...of`:

```
let s = 'Hello World!';
for (let char of s) {
  console.log(char);
  // Prints:
  // H
  // e
```

Using `for...of` we eliminate the need for a loop counter altogether, which has the added benefit that we'll never under- or over- shoot our collection's element list; we'll always loop across exactly the right number of elements within the given collection.

The `for...of` loop works with all collection types, from `String` to `Array` to `arguments` to `NodeList` (as well as newer collection types like `Map`, `Set`, etc.).

RegExp

A regular expression is a special string that describes a pattern to be used for matching or searching within other strings. They are also known as a *regex* or *regexp*, and in JavaScript we refer to `RegExp` when we mean the built-in `Object` type for creating and working with regular expressions.

You can think of regular expressions as a kind of mini programming language separate from JavaScript. They are not unique to JavaScript, and learning how to write and use them will be helpful in many other programming languages.

Even if you're not familiar with regular expression syntax (it takes some time to master), you've probably encountered similar ideas with wildcards. Consider the following Unix command:

```
ls *.txt
```

Here we ask for a listing of all files whose filename *ends with* the extension `.txt`. The `*` has a special meaning: *any character, and any number of characters*. Both `a.txt` and `file123.txt` would be matched against this pattern, since both end with `.txt`.

Regular expressions take the idea of defining patterns using characters like `*`, and extend it into a more powerful pattern matching language. Here's an example of a regular expression that could be used to match both common spellings of the word `"colour"` and `"color"`:

```
colou?r
```

The `?` means that the preceding character `u` is optional (it may or may not be there). Here's another example regular expression that could be used to match a string that starts with `id-` followed by 1, 2, or 3 digits (`id-1`, `id-12`, or `id-999`):

```
id-\d{1,3}
```

The `\d` means a digit (0-9) and the `{1,3}` portion means *at least one, and at most three*. Together we get *at least one digit, and at most three digits*.

There are many special characters to learn with regular expressions, which we'll slowly introduce.

Declaring JavaScript `RegExp`

Like `String` or `Array`, we can declare a `RegExp` using either a literal or the `RegExp` constructor:

```
let regex = /colou?r/; // regex literal uses /.../
let regex2 = new RegExp('colou?r');
```

Regular expressions can also have advanced search flags, which indicate how the search is supposed to be performed. These flags include `g` (globally match all occurrences vs. only matching once), `i` (ignore case when matching), and `m` (match across line breaks, multi-line matching) among others.

```
let regex = /pattern/gi; // find all matches (global) and ignore
case
let regex2 = new RegExp('pattern', 'gi'); // same thing using the
```

Understanding Regular Expression Patterns

Regular expressions are dense, and often easier to write than to read. It's helpful to use various tools to help you as you experiment with patterns, and try to understand and debug your own regular expressions:

- regexr.com
- Regulex
- regexpal.com

Matching Specific Characters

- `\ ^ $ . * + ? ( ) [ ] { } |` all have special meaning, and if you need to match them, you have to escape them with a leading `\`. For example: `\$` to match a `$`.

- Any other character will match itself. `abc` is a valid regular expression and means *match the letters abc*.

- The `.` means *any character*. For example `a.` would match `ab`, `a3`, or `a"`. If you need to match the `.` itself, make sure you escape it: `.\.` means *a period followed by any character*

- We specify a set of possible characters using `[]`. For example, if we wanted to match any vowel, we might do `[aeiou]`. This says *match any of the letters a, e, i, o, or u* and would match `a` but not `t`. We can also do the opposite, and define a negated set: `[^aeiou]` would match anything that is *not* a vowel. With regular expressions, it can often be easier to define

your patterns in terms of what they are not instead of what they are, since so many things are valid vs. a limited set of things that are not. We can also specify a range, `[a-d]` would match any of `a, b, c, d` but not `f, g` or `h`.

- Some sets are so common that we have shorthand notation. Consider the set of single digit numbers, `[0123456789]`. We can instead use `\d` which means the same thing. The inverse is `\D` (capital `D`), and means `[^0123456789]` (i.e., *not one of the digits*). If we wanted to match a number with three digits, we could use `\d\d\d`, which would match `123` or `678` or `000`.

- Another commonly needed pattern is *any letter or number* and is available with `\w`, meaning `[A-Za-z0-9_]` (all upper- and lower-case letters, digits 0 to 9, and the underscore). The inverse is available as `\W` and means `[^A-Za-z0-9_]` (everything *not* in the set of letters, numbers and underscore).

- Often we need to match blank whitespace (spaces, tabs, newlines, etc.). We can do that with `\s`, and the inverse `\S` (anything not a whitespace). For example, suppose we wanted to allow users to enter an id number with or without a space: `\d\d\d\s?\d\d\d` would match both `123456` and `123 456`.

- There are lots of other examples of pre-defined common patterns, such as `\n` (newline), `\r` (carriage return), `\t` (tab). Consult the MDN documentation for character classes to lookup others.

Define Character Matching Repetition

In addition to matching a single character or character class, we can also match sequences of them, and define *how many* times a pattern or match can/

must occur. We do this by adding extra information *after* our match pattern.

- `?` is used to indicate that we want to match something *once or none*. For example, if we want to match the word `dog` without an `s`, but also to allow `dogs` (with an `s`), we can do `dogs?`. The `?` follows the pattern (i.e., `s`) that it modifies, and indicates that it is optional.

- `*` is used when we want to match *zero or more* of something. `number \d*` would match `"number "` (no digits), `"number 1"` (one digit), and `"number 1234534123451334466600"`.

- `+` is similar to `*` but means *one or more*. `vroo+m` would match `"vroom"` but also `"vroooooooom"` and `"vroooooooooooooooooooooooooooooooom"`

- We can limit the number of matches to an exact number using `{n}`, which means *match exactly* `n` *times*. `vroo{3}m` would only match `"vroooom"`. We can further specify that we want a match to happen *match* `n` *or more times* using `{n,}`, or use `{n,m}` to indicate we want to match *at least `n` times and no more than `m` times: `\w{8,16}` would match 8 to 16 word characters, `"ABCD1234"` or `"zA5YncUI24T_3GHO"`

Define Positional Match Parameters or Alternatives

Normally the patterns we define are used to look *anywhere* within a string. However, sometimes it's important to specify *where* in the string a match is located. For example, we might care that an id number *begins* with some sequence of letters, or that a name doesn't *end* with some set of characters.

- `^` means start looking for the match at the *beginning* of the input string. We could test to see that a string begins with a capital letter like so: `^[A-Z]`.

- Similarly `$` means make sure that the match ends the string. If we wanted to test that string was a filename that ended with a period and a three letter extension, we could use: `\.\w{3}$` (an escaped period, followed by exactly 3 word characters, followed by the end of the string). This would match `"filename.txt"` but not `"filename.txt is a path"`.

- Sometimes we need to specify one of a number of possible alternatives. We do this with `|`, as in `red|green|blue` which would match any of the strings `"red"`, `"green"`, or `"blue"`.

Using `RegExp` with `String`s

So far we've discussed how to declare a `RegExp`, and also some of the basics of defining search patterns. Now we need to look at the different ways to use our regular expression objects to perform matches.

- `RegExp.test(string)` - used to test whether or not the given string matches the pattern described by the regular expression. If a match is made, returns `true`, otherwise `false`. `/id-\d\d\d/.test('id-123')` returns `true`, `/id-\d\d\d/.test('id-13b')` returns `false`.

- `String.match(regexp)` - used to find all matches of the given `RegExp` in the source `String`. These matches are returned as an `Array` of `String`s. For example, `'This sentence has 2 numbers in it, including the number 567'.match(/\d+/g)` will return the `Array` `['2', '567']` (notice the use of the `g` flag to find all matches globally).

- `String.replace(regexp, replacement)` - used to find all matches for the given `RegExp`, and returns a new `String` with those matches replaced by the replacement `String` provided. For example, `'50 ,`

`60,75.'.replace(/\s*,\s*/g, ', ')` would return `'50, 60, 75.'` with all whitespace normalized around the commas.

- `String.split(RegExp)` - used to break the given `String` into an `Array` of sub-strings, dividing them on the `RegExp` pattern. For example, `'one-two--three---four----five-----six'.split(/-+/)` would return `['one', 'two', 'three', 'four', 'five', 'six']`, with elements split on any number of dashes.

There are other methods you can call, and more advanced ways to extract data using RegExp, and you are encouraged to dig deeper into these concepts over time. Thinking about matching in terms of regular expressions takes practice, and often involves inverting your logic to narrow a set of possibilities into something you can define in code.

Practice Exercises

1. Write a function `log` that takes an `Array` of `String`s and displays them on the `console`.

2. An application uses an `Array` as a Stack (LIFO) to keep track of items in a user's shopping history. Every time they browse to an item, you want to `addItemToHistory(item)`. How would you implement this?

3. Write a function `buildArray` that takes two `Number`s, and returns an `Array` filled with all numbers between the given number: `buildArray(5, 10)` should return `[5, 6, 7, 8, 9, 10]`

4. Write a function `addDollars` that takes an `Array` of `Number`s and uses the array's `map()` method to create and return a new `Array` with each element having a `$` added to the front: `addDollars([1, 2, 3, 4])` should return `['$1', '$2', '$3', '$4']`

5. Write a function `tidy` that takes an `Array` of `String`s and uses the array's `map()` method to create and return a new `Array` with each element having all leading/trailing whitespace removed: `tidy([' hello', ' world '])` should return `['hello', 'world']`.

6. Write a function `measure` which takes an `Array` of `String`s and uses the array's `forEach()` method to determine the size of each string in the array, returning the total: `measure(['a', 'bc'])` should return `3`. Bonus: try to rewrite your code using the `Array`'s `reduce()` method.

7. Write a function `whereIsWaldo` that takes an `Array` of `String`s and uses the array's `forEach()` method to create a new `Array` with only the elements that contain the text `"waldo"` or `"Waldo"` somewhere in them:

`whereIsWaldo(['Jim Waldorf, 'Lynn Waldon', 'Frank Smith'])` should return `'Jim Waldorf, 'Lynn Waldon']`. Bonus: try to rewrite your code using the `Array`'s `filter()` method.

8. Write a function `checkAges` that takes two arguments: an `Array` of ages (`Number`); and a cut-off age (`Number`). Your function should return `true` if all of the ages in the `Array` are at least as old as the cut-off age: `checkAges([16, 18, 22, 32, 56], 19)` should return `false` and `checkAges([16, 18, 22, 32, 56], 6)` should return `true`. Bonus: try to rewrite your code using the `Array`'s `every()` method.

9. Write a function `containsBadWord` that takes two arguments: `badWords` (an `Array` of words that can't be used), and `userName` (a `String` entered by the user). Check to see if any of the words in `badWords` are contained within `userName`. Return the `badWord` that was found, or `null` if none are found.

10. A `String` contains a Key/Value pair separated by a `":"`. Using `String` methods, how would you extract the two parts? Make sure you also deal with any extra spaces. For example, all of the following should be considered the same: `"colour: blue"`, `"colour:blue"`, `"colour : blue"`, `"colour: blue "`. Bonus: how could you use a `RegExp` instead?

11. A `String` named `addresses` contains a list of street addresses. Some of the addresses use short forms: `"St."` instead of `"Street"` and `"Rd"` instead of `"Road"`. Using `String` methods, convert all these short forms to their full versions.

12. Room booking codes must take the following form: room number (`1-305`) followed by `-` followed by the month as a number (`1-12`) followed by the day as a number (`1-31`). For example, all of the following are valid:

`"1-1-1"`, `"250-10-3"`, `"66-12-12"`. Write a `RegExp` to check whether a room booking code is valid or not, which allows any of the valid forms.

13. Write a function that takes a `String` and checks whether or not it begins with one of `"Mr."`, `"Mrs."`, or `"Ms."`. Return `true` if it does, otherwise `false`. Bonus: try writing your solution using regular `String` methods *and* again as a `RegExp`.

14. Write a function that takes a password `String`, and validates it according to the following rules: must be between 8-32 characters in length; must contain one Capital Letter; must contain one Number; must contain one Symbol (`!@#$%^&*-+{}`). Return `true` if the given password is valid, otherwise `false`.

15. Write a `RegExp` for a Canadian Postal Code, for example `"M3J 3M6"`. Allow spaces or no spaces, capitals or lower case.

A Larger Problem Combining Everything:

You are asked to write JavaScript code to process a `String` which is in the form of a Comma-Separated Values (CSV) formatted data dump of user information. The data might look something like this:

```
0134134,John Smith,555-567-2341,62 inches
0134135   ,    June    Lee    ,  5554126347 ,        149 cm
0134136,       Kim Thomas       , 5324126347, 138cm`
```

Write a series of functions to accomplish the following, building a larger program as you go. You can begin with exercise.js:

1. Split the string into an `Array` of separate rows (i.e., an `Array` with rows separated by `\n`). Bonus: how could we deal with data that includes both Unix (`\n`) and Windows (`\r\n`) line endings?

2. Each row contains information user info: `ID`, `Name`, `Phone Number`, and `Height` info all separated by commas. Split each row into an `Array` with all of its different fields. You need to deal with extra and/or no whitespace between the commas.

3. Get rid of any extra spaces around the `Name` field

4. Using a `RegExp`, extract the Area Code from the `Phone Number` field. All `Phone Number`s are in one of two formats: `"555-555-5555"` or `"5555555555"`.

5. Check if the `Height` field has `"cm"` at the end. If it does, strip that out, convert the number to inches, and turn it into a `String` in the form `"xx inches"`. For example: `"152 cm"` should become `"59 inches"`.

6. After doing all of the above steps, create a new record with `ID`, `Name`, `Area Code`, `Height In Inches` and separate them with commas

7. Combine all these processed records into a new CSV formatted string, with rows separated by `\n`.

A sample solution is provided in solution.js.

Objects in JavaScript

So far we've been working with built-in `Objects` in JavaScript. We can also create our own in order to model complex data types in our programs. There are a number of ways to do this, and we'll look at a few of them now.

An `Object` in JavaScript is a *map* (also known as an *associative array* or a *dictionary*), which is a data structure composed of a collection of *key* and *value* pairs. We call an `Object`'s key/value pairs *properties*. Imagine a JavaScript `Object` as a dynamic "bag" of properties, a property-bag. Each *key* is a unique `String`, and an `Object` can only contain a given *key* once. An `Object` can have any number of *properties*, and they can be added and removed at runtime.

Much like we did with an `Array` or `RegExp`, we can create instances of `Objects` via literals. An `Object` literal always starts with `{` and ends with `}`. In between these curly braces we can optionally include a list of any properties (comma separated) we want to attach to this `Object` instance. These properties are written using a standard `key: value` style, with the property's name `String` coming first, followed by a `:`, then its value. The value can be any JavaScript value, including functions or other `Objects`.

Here are a few examples:

```
// an empty Object, with no properties
let o = {};

// a `person` Object, with one property, `name`
let person = { name: 'Tim Wu' };

// a `campus` Object, with `name` as well as co-ordinates (`lat`,
```

Accessing Elements in an Object

`Object` property names are `String`s, and we can refer to them either via the *dot operator* `.name`, or using the *bracket operator* `['name']` (similar to indexing in an `Array`):

```
let person = { name: 'Tim Wu' };

// get the value of the `name` property using the . operator
console.log(person.name);

// get the value of the `name` property using the [] operator
console.log(person['name']);
```

> Why would you choose the dot operator over the bracket operator, or vice versa? The dot operator is probably more commonly used; however, the bracket operator is useful in a number of scenarios. First, if you need to use a reserved JavaScript keyword for your property key, you'll need to refer to it as a string (e.g., `obj['for']`). Second, it's sometimes useful to be able to pass a variable in order to lookup a property value for a name that will be different at runtime. For example, if you are using usernames as keys, you might do `users[currentUsername]`, where `currentUsername` is a variable holding a `String` for the logged in user.

Destructuring Objects

In the same way that we *destructured* `Array` values into new variables, we can also use the same technique with an `Object`. Recall that JavaScript allows us to Destructuring Assignment to unpack values in an `Array` or `Object` into distinct variables. Consider each of the following methods, both of which

accomplish the same goal:

With an `Array`, we learned that you can *destructure* various elements into new variables:

```
// Co-ordinates for Seneca's Newnham Campus
let position = [43.796, -79.3486];

let [lat, lng] = position;
```

The same can be done with an `Object`. Imagine a complex `Object`, with lots of properties, but we're only interested in a few of them:

```
let senecaNewnham = {
  address: '1750 Finch Ave. East',
  city: 'Toronto',
  province: 'Ontario',
  postalCode: 'M2J 2X5',
  phoneNumber: '416.491.5050',
  lat: 43.796,
  lng: -79.3486,
};

// Destructure only the `lat` and `lng` properties
let { lat, lng } = senecaNewnham;
```

This is a powerful technique for extracting data from an `Object`.

Modifying Object Properties

`Object` literals allow us to define an initial set of properties on an `Object`, but we aren't limited to that set. We can easily add new ones:

```
let data = {};

data.score = 17;
data.level = 3;
data.health = '***';
```

Here we define an empty `Object`, but then add new properties. Because we can add properties after an `Object` is created, we always have to deal with a property not existing. If we try to access a property that does not exist on an `Object`, there won't be an error. Instead, we will get back `undefined` for its value:

```
let currentScore = data.score; // `score` exists on `data`, and we get back the value `17`
let inventory = data.inventory; // `inventory` does not exist on `data`, so we get back `undefined`
```

Because properties may or may not exist at runtime, it's important to always check for a value before trying to use it. We could rewrite the above to first check if `data` has an `inventory` property:

```
if (data.inventory) {
  // `data` has a value for `inventory`, use data.inventory here...
} else {
  // there is no `inventory` on `data`, do something else...
}
```

Another common situation where you have to deal with this is working with deep structures. Consider an `Object` that defines the structure of a level in a video game. The level includes various `rooms`, some of which contain a

`monster`:

```
let gameLevel = {
  name: 'Level 1',
  rooms: {
    // Each room has a unique ID
    R31343: {
      name: 'Front Hallway',
    },
    R31344: {
      name: 'Kitchen',
      monster: {
        name: 'Bear',
        strength: 15,
      },
    },
    R31345: {
      name: 'Back Hallway',
    },
    R31346: {
      name: 'Sitting Room',
      monster: {
        name: 'Dog',
        strength: 8,
      },
    },
  },
};
```

When working this code, we can access a particular room by its `ID`:

```
// Get a reference to the Kitchen
let room = gameLevel.rooms.R31344;
```

However, we used an `ID` that doesn't exist, we'd get back `undefined`:

```
// Get a reference to the TV Room (no such room!)
let room = gameLevel.rooms.R31347; // <-- room is `undefined`
```

If we then try to access the `monster` in that room, our program will crash:

```
let room = gameLevel.rooms.R31347; // <-- room is `undefined`
console.log(room.monster); // <-- crash! room is `undefined` so we
can't access `monster within it
```

JavaScript provides a few ways to deal with this problem. Consider:

```
let room = gameLevel.rooms.R31347;

// Version 1
if (room) {
  // only access room if it is truthy
}

// Version 2
if (room && room.monster) {
  // only try to get .monster if room is truthy
}

// Version 3
if (room?.monster) {
  // same as 2, but using ?. syntax
}
```

In the third version above we've used optional chaining via the `?.` operator. This stops us from going any further in an object chain, when something is undefined.

Using Objects: dealing with optional parameters

A very common pattern in JavaScript programs that uses this concept is optional argument passing to functions. Instead of using an unknown number of `arguments` for a function, we often use an `options` `Object`, which may contain values to be used in the function. Consider the case of starting a game and sometimes passing existing user data:

```
// Make sure `options` exists, and use an empty `Object` instead
if it's missing.
// If we don't do this, we'll get an error if we try to do
`options.score`, since
// we can't lookup the `score` property on `undefined`.
function initGame(options = {}) {
  // If the user already has a score, use that, otherwise default
to 0
  let score = options.score || 0;
  // If the user is already on a level, use that, otherwise
default to 1
  let level = options.level || 1;
  // If the user has collected an items in her inventory, use
that, otherwise an empty Array
  let inventory = options.inventory || [];

  // Begin the game, passing the values we have determined above
  playGame(score, level, inventory);
}

// Define our options: we have a score and level, but no inventory
let options = {
  score: 25,
```

In the code above, we have an `options` `Object` that defines some, but not all of the properties our `initGame` function might use. We wrote `initGame` using a single argument so that it was easier to call: we didn't need to worry about the order or number of arguments, and could instead just define an `Object` with all of the properties we had. The `initGame` function examined the `options` at runtime to see which properties existed, and which were `undefined` and needed a default value instead. Recall that we can use the *logical OR* (`||`) operator to choose between two values at runtime.

It's also common to see people use *destructuring* here:

```
function processStudent(student) {
  let { name, studentId, username, email } = student;
  // Use values destructured from student object
}

processStudent({
  name: 'Tim Wu',
  studentId: '10341346',
  username: 'timw',
  email: 'timw@myseneca.ca',
});
```

The value of what we've done above is that passing many arguments to a function is easier when we can name them as properties on an `Object` instead of having to pass them positionally as arguments.

Updating, Clearing, and Removing properties

We've seen that properties can be defined when declared as part of a literal

and added later via the `.` or `[]` operators. We can also update or remove values after they are created:

```
let o = {};

// Add a name property
o.name = 'Tim Wu';

// Update the name property to a new value, removing the old one.
o.name = 'Mr. Timothy Wu';
```

An `Object`'s property keys are unique, and setting a value for `o.name` more than once doesn't add more properties--it overwrites the value already stored in the existing property. We can also *clear* (remove the value but not the key) or *delete* (remove the entire property from the object, key and value) things from an `Object`.

```
let o = {};

// Add a `height` property
o.height = '35 inches';

// Add an owner ID property
o.owner = '012341341';

// Clear the value of `height`. We leave the `height` key, but get
rid of the '35 inches' value
o.height = null;

// Completely remove the owner property from the object (both the
key and its value)
delete o.owner;
```

> Why would you choose to assign `null` vs. use `delete`? Often we want to

> get rid of a key's value, but will use the key again in the future (e.g., add a new value). In such cases we just *null the value* by assigning the key a value of `null`. If we know that we'll never use this key again, and we don't want to retain it on the `Object`, we can instead completely remove the property (key and value) with `delete`. You'll see both used. For the most part, setting a key's value to `null` is probably what you want.

Using Objects: creating sets to track arbitrary lists

Another common use of `Object`s, and their unique property keys, is to keep track of a sets, for example to count or keep track of an unknown number of items. Consider the following program, which tracks how many times each character appears within a `String`. The code uses the `[]` operator to allow for the keys to be created and accessed via a variable (`char`). Without an `Object` we would have to hard-code variables for each separate letter.

```
// An empty `Object`, which we'll populate with keys (letters) and
values (counts)
let characterCounts = {};

let sentence = 'The quick brown fox jumped over the lazy dog.';
let char;
let count;

// Loop through all characters in sentence
for (let char of sentence) {
  // Get the current count for this character, or use 0 if we
haven't seen it before
  count = characterCounts[char] || 0;
  // Increase the count by 1, and store it in our object
```

Complex Property Types: `Object`, `Function`

We said earlier that `Object` properties can be any valid JavaScript type. That includes `Number`, `String`, `Boolean`, etc., also `Object` and `Function`. A property may define a complex `Object` of its own:

```
let part = {
  id: 5,
  info: {
    name: 'inner gasket',
    shelf: 56713,
    ref: [5618, 5693],
  },
};
```

Here we define a `part`, which has an id (`part.id`) as well as a complex property named `info`, which is itself an `Object`. We access properties deep in an `Object` the same way as a simple property, for example: `part.info.ref.length` means: get the `length` of the `ref` array on the `info` property of the `part` `Object`. An `Object`'s properties can be `Object`s many levels deep, and we use the `.` or `[]` operators to access these child properties.

An `Object` property can also be a function. We call these functions *methods*. A *method* has access to other properties on the `Object` via the `this` keyword, which refers to the current `Object` instance itself. Let's add a `toString()` method to our `part` `Object` above:

```
let part = {
  id: 5,
  info: {
    name: 'inner gasket',
    shelf: 56713,
    ref: [5618, 5693],
  },
  toString: function () {
    return `${this.info.name} (#${this.id})`;
  },
};

console.log(part.toString()); // prints "inner gasket (#5)" to the
console.
```

The `toString` property is just like any other key we've added previously, except its value is an *anonymous function*. Just as we previously bound function expressions to variables, here a function expression is bound to an `Object`'s property. When we write `part.toString` we are accessing the function stored at this key, and by adding the `()` operator, we can invoke it: `part.toString()` says *get the function stored at part.toString and call it*. Our function accesses other properties on the `part` `Object` by using `this.*` instead of `part.*`. When the function is run, `this` will be the same as `part` (i.e., a reference to *this* `Object` instance).

> The `this` keyword in JavaScript is used in different contexts, and has a different meaning depending on where and how it is used. We will return to `this` and its various meanings throughout the course.

Suggested Readings

- Object-oriented JavaScript for beginners

- ExploringJS, Chapter 17. Objects and Inheritance
- ExploringJS, Chapter 20. Dates
- ExploringJS, Chapter 21. Math

Constructor Functions

Sometimes we need to create lots of `Objects` that have the same layout. For example, we might be defining lots of users in an application. All of our user `Objects` need to work the same way so that we can pass them around within our program, to and from functions. Every `user` needs to have the same set of properties and methods, so we decide to write a factory function that can build our `user` `Objects` for us based on some data. We call such functions a `Constructor`:

```
// Define a Constructor function, `User`
function User(id, name) {
  // Attach the id to an Object referenced by `this`
  this.id = id;
  // Attach the name to an Object referenced by `this`
  this.name = name;
}

// Create a new instance of a User (Object)
let user1 = new User(1, 'Sam Smith');
// Create another new instance of a User (Object)
let user2 = new User(2, 'Joan Winston');
```

Notice that unlike all previous functions we've defined, the `User` function starts with a capital `U` instead of a lower case `u`. We use this naming convention to indicate that `User` is special: a constructor function. A constructor function needs to be called with the extra `new` keyword in front of it. When we say `new User(...)` we are saying, *create a new object, and pass it along to User so it can attach various things to it*.

A constructor can also add methods to an object via `this`:

```
// Define a Constructor function, `User`
function User(id, name) {
  this.id = id;
  this.name = name;

  // Add a toString method
  this.toString = function () {
    return `${this.name} (#${this.id})`;
  };
}

// Create a new instance of a User (Object)
let user1 = new User(1, 'Sam Smith');
console.log(user1.toString()); // 'Sam Smith (#1)
```

In the code above, we're creating a new function every time we create a new User. As we start to create lots of users, we'll also be creating lots of duplicate functions. This will cause our program to use more and more resources (memory), which can lead to issues as the program scales.

Object Prototypes

What we would really like is a way to separate the parts of a User that are different for each user (the data: `id`, `name`), but somehow share the parts that are the same (the methods: `toString`). JavaScript gives us a way to accomplish this via an `Object`'s `prototype`.

JavaScript is unique among programming languages in the way it accomplishes sharing between `Object`s. All object-oriented languages provide some mechanism for us to share or inherit things like methods in a type hierarchy. For example, C++ and Java use classes, which can inherit from one another to define methods on parents vs. children. JavaScript uses *prototypal inheritance*

and a special property called `prototype`.

In JavaScript, we always talk about `Object`s, because every object is an instance of `Object`. Notice the capital `O` in `Object`, which should give you an indication of what it is: a constructor function. In a previous week we said that an `Array` is an `Object`, and a `RegExp` is an `Object`. This is true because of JavaScript's type system, where almost everything is *chained* to `Object`.

JavaScript objects always have a prototype, which is an object to which their `.prototype` property refers. At runtime, when we refer to an object's property, JavaScript first looks for that property on the object itself. If it doesn't find it, the prototype object is visited, and the same search is done. The process continues until the end of the prototype chain is reached at `Object`.

Let's rewrite our `User` so that the `toString` method is moved from each user instance to the prototype of all user instances:

```
// Define a Constructor function, `User`
function User(id, name) {
  this.id = id;
  this.name = name;
}

User.prototype.toString = function () {
  return `${this.name} (#${this.id})`;
};
```

This code looks very similar to what we originally wrote. Notice that we've moved `toString` out of the `User` function, and instead attached it to `User.prototype`. By doing so, we'll only ever need a single copy of this function: every `new User()` instance we create will also include a reference to a prototype object, which contains our function. When we use

`user1.toString()`, JavaScript will do something like this:

1. does `user1` have a property called `toString`? No, we didn't add one in the constructor.
2. does `user1.prototype` have a property called `toString`? Yes, use that.

What if we'd written `user1.something()`?

1. does `user1` have a property called `something`? No, we didn't add one in the constructor.
2. does `user1.prototype` have a property called `something`? No.
3. does `user1.prototype.prototype` (i.e., `Object`) have a property called `something`? No.
4. there are no more objects in the prototype chain, throw an error

```
user1.something();
// TypeError: user1.something is not a function
```

Whenever a method is used on a prototype, we still pass the current instance so we can get access to its data. Notice in our `User.prototype.toString` method, we still referred to `this`, which will be the instance of our user, and give us access to the correct data (`name`, `id`).

> There are times when defining a method inside a constructor makes sense vs. putting it on the prototype. The prototype will only have access to *public properties* of an object instance, meaning things you explicitly add to `this` and expose to the rest of your program. Sometimes we want to define some data, but *hide* it from the rest of a program, so it can't be changed after it gets created. Consider the following example, which uses a *closure* to retain access to a variable in the scope of the constructor without exposing it:

```
function User(id, name) {
  this.id = id;
  this.name = name;

  // private variable within User function, not attached to
`this`.
  // Normally this variable would go out of scope after User()
completed;
  // however, we will use a closure function below to capture this
scope.
  let createdAt = Date.now();

  // Return the number of ms this player has been playing
  this.playerAgeMS = function () {
    let currentTime = Date.now();

    // Access `createdAt` in the parent scope, which we retain via
this closure function.
    // Calculate how many ms between createdAt and the current
time.
    return currentTime - createdAt + ' ms';
  };
}

let user = new User(1, 'Tom');
// We can access the total time this player has existed, but not
modify it.
console.log(user.playerAgeMS());
// displays "4183 ms"
console.log(user.playerAgeMS());
// displays "5287 ms"
```

JavaScript's `class` and `Object`

For a long time, JavaScript didn't have any notion of a class. Most Object-Oriented languages are based on the idea of a class, but JavaScript only has runtime instances (i.e., `Object`s) and didn't need them.

In recent years, a new syntax has been added to JavaScript to allow those more familiar with traditional OOP style programming to define their `Object`s using a new `class` keyword.

Let's recreate our code above as a `class` in JavaScript:

```
class User {
  id;
  name;

  constructor(id, name) {
    this.id = id;
    this.name = name;
  }

  toString() {
    return `${this.name} (#${this.id})`;
  }
}
```

This code still uses the same prototype technique we learned above above, but does so in a more familiar syntax.

We can even use other OOP features like inheritance:

```
class Student extends User {
```

Practice Exercise

Morse Code translator

Morse code is a system of encoding developed in the 1800s that allowed transmission of textual messages over signal systems that only supported on/off (1 and 0) notations.

Complete the program below as specified. Your program should be able to translate messages like `-- --- .-. ... ./-.-. --- -.. .` into `MORSE CODE` and vice versa. Use what you learned above about `Object`s, and also some of the built-in `Object`s we've studied, in particular `RegExp` and `String`.

Use the following limited set of morse code to use in this exercise. You could expand your program to handle more complex messages later if you want:

Letter	Morse
A	`.-`
B	`-...`
C	`-.-.`
D	`-..`
E	`.`
F	`..-.`

Letter	Morse
G	`--.`
H	`....`
I	`..`
J	`.---`
K	`-.-`
L	`.-..`
M	`--`
N	`-.`
O	`---`
P	`.--.`
Q	`--.-`
R	`.-.`
S	`...`
T	`-`

Letter	Morse
U	`..-`
V	`...-`
W	`.--`
X	`-..-`
Y	`-.--`
Z	`--..`
space	`/`

NOTE: letters are separated by a single space (`' '`) within a word, and words are separated with a `/`. For example, the words `MORSE CODE` would translate to `-- --- .-. ... ./-.-. --- -.. .`

```
// Object to provide lookup of morse code (value) for a given
letter (key).
let alpha = {
  // define the mapping here as a literal
};

// Object to provide lookup of letter (value) for a given morse
code (key).
let morse = {};
// Hint: use the [] operator to specify these special key values
rather than a literal.
```

You can download the code above as well as a possible solution.

Running a Development Web Environment

Developing for the web requires at least 3 things pieces of software:

1. a proper code editor which, is aware of HTML, JavaScript, and CSS
2. a web client (i.e., browser), with developer and debugging tools
3. a web server, to serve your web pages over HTTP to a browser

Code Editor

For our code editor, we will be using Visual Studio Code, which is a free (open source) code editor created and maintained by Microsoft. It also works on Windows, macOS, and Linux. Make sure you have downloaded and installed it on all the computers you will use for web development.

Web Client

For our web client we will use the many web browsers we introduced in Week 1, namely:

- Google Chrome for desktop and Android
- Microsoft Edge and Internet Explorer (IE)
- Apple Safari and Safari for iOS
- Mozilla Firefox

- Opera

There are many more, and you are highly encouraged to install as many as possible.

Web Server

We will also need a **web server** to host our web pages and applications. Installing and running a web server can be complicated. Industry-grade web servers like Apache and nginx are free and can be installed and run on your local computer; however, they are much more complicated and powerful than anything we will need for hosting our initial web pages.

For our purposes, we will use one of the many simple node.js based HTTP servers. In order to use them, do the following:

1. Make sure you have installed node.js on your computer.
2. In a terminal window, navigate to the directory that you want your web server to host. For example `cd my-website`
3. Now download and run a web server using the npx command.

For example, you can use the serve web server like this:

```
cd my-website
npx serve
Need to install the following packages:
  serve@14.2.1
Ok to proceed? (y)

   ┌──────────────────────────────────────────┐
   │                                          │
   │   Serving!                               │
```

You can now open your web browser to `http://localhost:3000` and browser your files. This uses the `http` protocol, and connects you to the special IP address `127.0.0.1`, also known as localhost (i.e., you can also use `http://localhost:3000`). The localhost IP address always refers to *this* computer, and allows you to connect network clients to your own machine. The final `:3000` portion of the URL is a port number. Together, `http://127.0.0.1:3000` means *connect using HTTP to my local computer on port 3000.*

NOTE: the second External IP address will be different than the above, but 127.0.0.1 will always be correct.

When you are done testing your web site, stop the web server by pressing `CTRL-C` in your terminal window. To run the server again, use `npx serve`.

Suggested Readings

- HTML: HyperText Markup Language on MDN
- HTML Basics
- Learning HTML: Guides and Tutorials
- HTML Reference

HTML

HTML is the HyperText Markup Language. It allows us to write *content* in a document, just as we would in a file created by a word processor. Unlike a regular text file, it also includes structural and layout information about this content. We literally *mark up* the text of our document with extra information.

When talking about HTML's markup, we'll often refer to the following terms:

- content: any text content you want to include can usually be written as-is.
- tag: separated from regular content, tags are special text (names) wrapped in `<` and `>` characters, for example the paragraph tab `<p>` or the image tag `<img>`.
- element: everything from the beginning of an opening tag to the closing tag, for example: `<h1>Chapter 1</h1>`. Here an element is made up of an `<h1>` tag (i.e., opening Heading 1 tag), the text content `Chapter 1`, and a closing `</h1>` tag. These three components taken together create an `h1` element in the document.
- attribute: optional characteristics of an element defined using the style `name` or `name="value"`, for example `<p id="error-message" hidden>There was an error downloading the file</p>`. Here two attributes are included with the `p` element: an `id` with value `"error-message"` (in quotes), and the `hidden` attribute (note: not all attributes need to have a value). Full list of common attributes.
- entity: special text that should not be confused for HTML markup. Entities begin with `&` and end with `;`. For example, if you need to use the `<` character in your document, you need to use `<` instead, since `<` would be interpreted as part of an HTML tag. ` ` is a single whitespace and `&` is the `&` symbol. Full list of named entities.

HTML Document

The first HTML page ever created was built by Tim Berners-Lee on August 6, 1991.

Since then, the web has gone through many versions:

- HTML - created in 1990 and standardized in 1997 as HTML 4
- xHTML - a rewrite of HTML using XML in 2000
- HTML5 - the current standard.

Basic HTML5 Document

Here's a basic HTML5 web page:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>My Web Page</title>
  </head>

  <body>
    <!-- This is a comment -->
    <h1>Hello World!</h1>
  </body>
</html>
```

Let's break this down and look at what's happening.

1. `<!doctype html>` tells the browser what kind of document this is (HTML5),

and how to interpret/render it

2. `<html>` the root element of our document: all other elements will be included within `<html>...</html>`.
3. `<head>` provides various information *about* the document as opposed to providing its content. This is metadata that describes the document to search engines, web browsers, and other tools.
4. `<meta>` an example of metadata, in this case defining the character set used in the document: utf-8
5. `<title>` an example of a specific (named) metadata element: the document's title, shown in the browser's title bar. There are a number of specific named metadata elements like this.
6. `<body>` the content of the document is contained within `<body>...</body>`.
7. `<!-- ... -->` a comment, similar to using `/* ... */` in C or JavaScript
8. `<h1>` a heading element (there are headings 1 through 6), which is a title or sub-title in a document.

Now let's try creating and loading this file in our browser:

1. Make a directory on your computer called `my-website`
2. Create a new file in `my-website` named `index.html` (the `index.html` name is important, as it represents the main entry point to a directory of HTML files and other web resources)
3. Use Visual Studio Code to open your `my-website/index.html` file
4. Copy the HTML we just discussed above, and paste it into your editor
5. Save your `index.html` file
6. In a terminal, navigate to your `my-website` directory
7. Start a web server by typing `npx serve` (you must do this from **within** the `my-website` directory)

8. Open your web browser (Chrome, Firefox, etc) and enter `http://localhost:3000` in the URL bar
9. Make sure you can see a new page with `Hello World!` in black text.

Now let's make a change to our document:

1. Go back to your editor and change the `index.html` file so that instead of `Hello World!` you have `This is my web page.`
2. Save your `index.html` file.
3. Go back to your browser and hit the **Refresh** button.
4. Make sure your web page now says `This is my web page.`

Every time we update anything in our web page, we have to refresh the web page in our browser. The web server will serve the most recent version of the file on disk when it is requested. Web browsers and servers disconnect from one another after processing a request/response.

Common HTML Elements

There are many HTML elements you'll learn and use, but the following is a good initial set to get you started.

You can see an example page that uses every HTML element here.

Metadata

Information *about* the document vs. the document's content goes in various metadata elements:

- `<link>` - links from this document to external resources, such as CSS stylesheets

- `<meta>` - metadata that can't be included via other elements
- `<title>` - the document's title

Major Document Sections

- `<html>` - the document's root element, containing all other elements
- `<head>` - machine-readable metadata about the document
- `<body>` - the document's content

Content Sections

These are organizational blocks within the document, helping give structure to the content and provide clues to browsers, screen readers, and other software about how to present the content:

- `<header>` - introductory material at the top of a document
- `<nav>` - content related to navigation (a menu, index, links, etc)
- `<main>` - the main content of the document. For example, a news article's paragraphs vs. ads, links, navigation buttons, etc.
- `<h1>, <h2>, ..., <h6>` - (sub) headers for different sections of content
- `<footer>` - end material (author, copyright, links)

Text Content

We organize content into "boxes," some of which have unique layout characteristics.

- `<div>` - a generic container we use to attach CSS styles to a particular area of content
- `<ol>` - an ordered list (1, 2, 3) of list items

- `<ul>` - an unordered list (bullets) of list items
- `<li>` - a list item in an `<ul>` or `<ol>`
- `<p>` - a paragraph
- `<blockquote>` - an extended quotation

Inline Text

We also use elements within larger text content to indicate that certain words or phrases are to be shown differently:

- `<a>` - an "anchor" element, which will produce a hyperlink, allowing users to click and navigate to some other document.
- `<code>` - formats the text as computer code vs. regular text.
- `<em>` - adds emphasis to the text (often in italics)
- `<span>` - another generic container, used to define CSS styles

Multimedia

In addition to text, HTML5 also defines a number of rich media elements:

- `<img>` - an element used to embed images in a document.
- `<audio>` - an element used to embed sound in a document.
- `<video>` - an element used to embed video in a document
- `<canvas>` - a graphical area (rectangle) used to draw with either 2D or 3D using JavaScript.

Scripting

We create dynamic web content and applications through the use of scripting:

- `<script>` - used to embed executable code in a document, typically

JavaScript.

Practical Examples

- Lists: ordered and unordered
- Anchors: creating hyperlinks
- Images: using img
- Text: text sections

HTML Elements

HTML Element Types: Block vs. Inline

Visual HTML elements are categorized into one of two groups:

1. Block-level elements: create a "block" of content in a page, with an empty line before and after them. Block elements fill the width of their parent element. Block elements can contain other block elements, inline elements, or text.
2. Inline elements: creates "inline" content, which is part of the containing block. Inline elements can contain other inline elements or text.

Consider the following HTML content:

```
<body>
  <p>The <em>cow</em> jumped over the <b>moon</b>.</p>
</body>
```

Here we have a `<p>` paragraph element. Because it is a block-level element, this paragraph will fill its container (in this case the `<body>` element). It will also have empty space added above and below it.

Within this block, we also encounter a number of other inline elements. First, we have simple text. However, we also see the `<em>` and `<b>` elements being used. These will affect their content, but not create a new block; rather, they will continue to flow inline in their container (the `<p>` element).

Empty Elements

Many of the elements we've seen so far begin with an opening tag, and end with a closing tag: `<body></body>`. However, not all elements need to be closed. Some elements have no *content*, and therefore don't need to have a closing tag. We call these empty elements.

An example is the `<br>` line break element. We use a `<br>` when we want to tell the browser to insert a newline (similar to using `\n` in C):

```
<p>Knock, Knock<br />Who's there?</p>
```

Other examples of empty elements include `<hr>` (for a horizontal line), `<meta>` for including metadata in the `<head>`, and a dozen others.

Grouping Elements

Often we need to group elements in our page together. We have a number of pre-defined element container options for how to achieve this, depending on what kind of content we are creating, and where it is in the document.

Using this so-called semantic markup helps the browser and other tools (e.g., accessibility) determine important structural information about the document (see this post for a great discussion):

- `<header>` - introductory material at the top of a
- `<nav>` - content related to navigation (a menu, index, links, etc)
- `<main>` - the main content of the document.

- `<article>` - a self-contained composition, such as a blog post, article, etc.
- `<section>` - a group of related elements in a document representing one section of a whole
- `<footer>` - end material (author, copyright, links)

Sometimes there is no appropriate semantic container element for our content, and we need something more generic. In such cases we have two options:

- `<div>` - a generic block-level container
- `<span>` - a generic inline container

```
<div>
  <p>
    This is an example of a using a div element. It also includes
this
    <span><em>span</em> element</span>.
  </p>
  <p>
    Later we'll use a div or span like this to target content in
our page with JavaScript or CSS
    styles.
  </p>
  <p></p>
</div>
```

Tables

Sometimes our data is tabular in nature, and we need to present it in a grid. A number of elements are used to create them:

- `<table>` - the root of a table in HTML
- `<caption>` - the optional title (or caption) of the table

- `<thead>` - row(s) at the top of the table (header row or rows)
- `<tbody>` - rows that form the main body of the table (the table's content rows)
- `<tfoot>` - row(s) at the bottom of the table (footer row or rows)

We define rows and columns of data within the above using the following:

- `<tr>` - a single row in a table
- `<td>` - a single cell (row/column intersection) that contains table data
- `<th>` - a header (e.g., a title for a column)

We can use the `rowspan` and `colspan` attributes to extend table elements beyond their usual bounds, for example: have an element span two columns (`colspan="2"`) or have a heading span 3 rows (`rowspan="3"`).

```
<table>
  <caption>
    Order Information
  </caption>

  <thead>
    <tr>
      <th>Quantity</th>
      <th>Colour</th>
      <th>Price (CAD)</th>
    </tr>
  </thead>

  <tbody>
    <tr>
      <td>1</td>
      <td>Red</td>
      <td>$5.60</td>
```

Suggested Readings

- HTML Tables (MDN)
- Images in HTML (MDN)
- Video and Audio Content (MDN)
- HTML Reference

Multimedia

Images, Audio & Video

HTML5 has built in support for including images, videos, and audio along with text. We specify the media source we want to use, and also how to present it to the user via different elements and attributes

```
<!-- External image URL, use full width of browser window -->
<img
  src="https://images.unsplash.com/
photo-1502720433255-614171a1835e?ixlib=rb-0.3.5&ixid=eyJhcHBfaWQiOjEyMDd9&s=344dfca9dc8cb137a4b1c2c711752bc5"
/>

<!-- Local file cat.jpg, limit to 400 pixels wide -->
<img src="cat.jpg" alt="Picture of a cat" width="400" />
```

HTML5 has also recently added the `<picture>` element , to allow for an optimal image type to be chosen from amongst a list of several options.

We can also include sounds, music, or other audio:

```
<!-- No controls, music will just auto-play in the background. Only MP3 source provided -->
<audio
  src="https://ia800607.us.archive.org/15/items/music_for_programming/music_for_programming_1-datassette.mp3"
  autoplay
></audio>

<!-- Audio with controls showing, multiple formats available -->
<audio controls>
  <source src="song.mp3" type="audio/mp3" />
  <source src="song.ogg" type="audio/ogg" />
  <p>
    Sorry, your browser doesn't support HTML5 audio. Here is a
    <a href="song.mp3">link to the audio</a> instead
  </p>
  .
</audio>
```

Including video is very similar to audio:

```
<!-- External Video File, MP4 file format, show controls -->
<video
  src="http://commondatastorage.googleapis.com/gtv-videos-bucket/sample/BigBuckBunny.mp4"
  controls
></video>

<!-- Local video file in various formats, show with controls -->
<video width="320" height="240" controls>
  <source src="video.mp4" type="video/mp4" />
  <source src="video.ogg" type="video/ogg" />
  <source src="video.webm" type="video/webm" />
  <p>Sorry, your browser doesn't support HTML5 video</p>
</video>
```

NOTE: the `<audio>` and `<video>` elements must use source URLs that point to actual audio or video files and not to a YouTube URL or some other source that is actually an HTML page.

Including Scripts

We've spent a good portion of the course learning about JavaScript. So far, all of our code has been written in a stand-alone form, executed in the Firefox Scratchpad, or by using node.js.

Our ultimate goal is to be able to run our JavaScript programs within web pages and applications. To do that, we need a way to include JavaScript code in an HTML file. Obviously HTML isn't anything like JavaScript, so we can't simply type our code in the middle of an HTML file and expect the browser to understand it.

Instead, we need an HTML element that can be used to contain (or link to) our JavaScript code. HTML provides such an element in the form of the `<script>` element.

We can use `<script>` in one of two ways.

Inline Scripts

First, we can embed our JavaScript program directly within the content area of a `<script>` element:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Web Page with Script</title>
  </head>

  <body>
    <script>
      console.log('Hello World!');
    </script>
  </body>
</html>
```

Such `<script>` elements can occur anywhere in your HTML, though it is common to put them at the end of the `<body>`. We can also include more than one, and each shares a common global environment, which is useful for combining scripts:

```
<script>
  // Define a global variable `msg` with a String
  var msg = 'Hello World!';
</script>

<script>
  // Access the global variable `msg`, defined in another <script>, but within the same JS environment
  console.log(msg);
</script>
```

External Scripts Linked via URL

As our JavaScript programs get larger, embedding them directly within the HTML file via an inline `<script>` starts to become unwieldy. For very small scripts, and debugging or experimentation, inline scripts are fine. However, HTML and JavaScript aren't the same thing, and it's useful to separate them into their own files for a number of reasons.

First, browsers can cache files to improve load times on a web site. If you embed a large JavaScript file in the HTML, it can't be cached.

Second, your HTML becomes harder to read. Instead of looking at semantic content about the structure of your page, now you have script mixed in too. This can make it harder to understand what you're looking at while debugging.

Third, there are lots of tools for HTML, and even more for JavaScript, that only work when fed the proper file type. For example, we often use linters or bundling tools in JavaScript. We can't do that if our JavaScript is combined with HTML markup.

For these and other reasons, it's common to move your JavaScript programs to separate files with a `.js` file extension. We then tell the browser to load and run these files as needed via our `<script>` tag like so:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Web Page with Script</title>
  </head>

  <body>
    <script src="script.js"></script>
  </body>
</html>
```

In this case, we have no content within our `<script>` element, and instead include a `src="script.js"` attribute. Much like the `<img>` element, a `<script>` can include a `src` URL to load at runtime. The browser will begin by loading your `.html` file, and when it encounters the `<script src="script.js">` element, it will begin to download `script.js` from the web server, and then run the program it contains.

We can combine both of these methods, and include as many scripts as we need. The scripts we include can be:

- embedded inline in the HTML
- a relative URL to the same web server that served the HTML file
- an absolute URL to another web server somewhere else on the web

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Web Page with Scripts</title>
  </head>

  <body>
    <script src="https://scripts.com/some/other/external/script/file.js"></script>
    <script src="local-script.js"></script>
    <script>
      // Use functions and Objects defined in the previous two files
      doSomethingAmazing();
    </script>
  </body>
</html>
```

Validating HTML

It's clear that learning to write proper and correct HTML is going to take practice. There are lots of elements to get used to, and learn to use in conjunction. Also each has various attributes that have to be taken into account.

Browsers are fairly liberal in what they will accept in the way of HTML. Even if an HTML file isn't 100% perfect, a browser can often still render something. That said, it's best if we do our best to provide valid HTML.

In order to make sure that your HTML is valid, you can use an HTML Validator. There are a few available online:

- https://html5.validator.nu/
- https://validator.w3.org/

Both allow you to enter a URL to an existing web page, or enter HTML directly in a text field. They will then attempt to parse your HTML and report back on any errors or warnings, for example: an element missing a closing tag.

Practical Examples

- Image: img with fixed width
- Table: table with multiple rows, columns
- Audio: single audio source
- Video: single video source

DOM Introduction

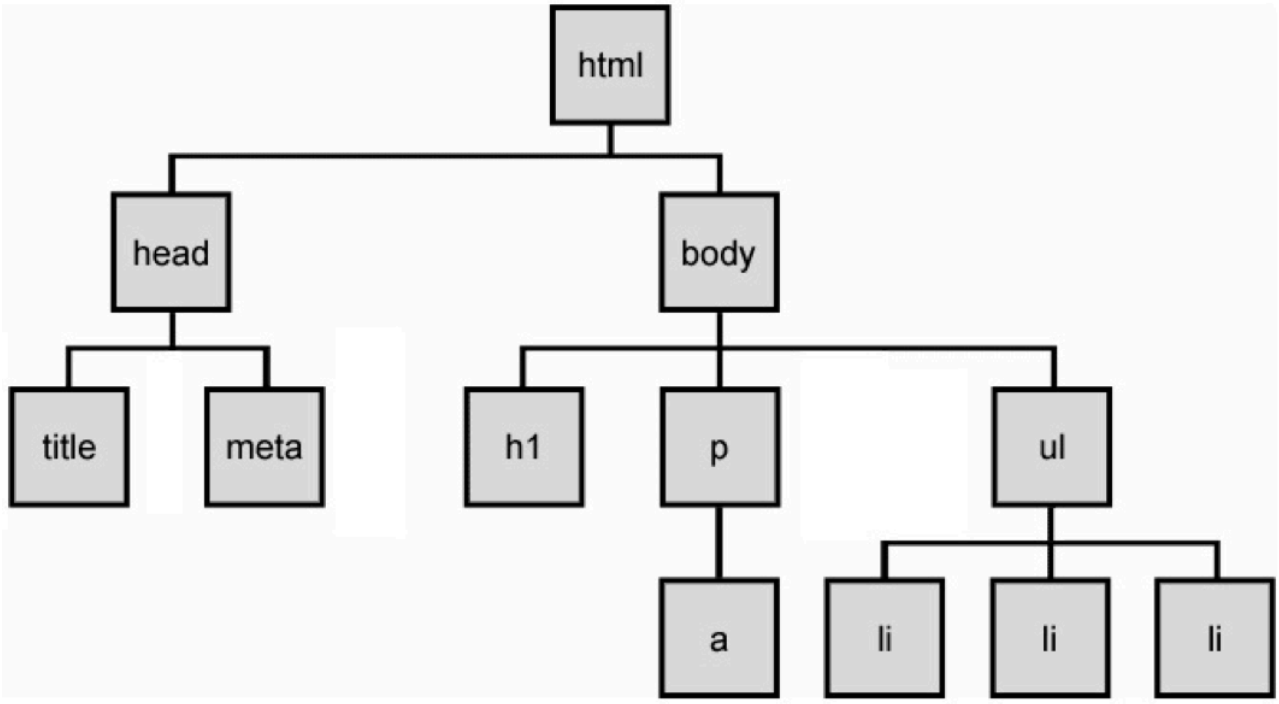
From HTML to the DOM

Web pages rely on HTML for their initial structure and content. We write web pages using HTML, and then use web browsers to parse and render that HTML into a living (i.e., modifiable at runtime) tree structure. Consider the following HTML web page:

The DOM Tree is a living version of our HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a Document!</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Welcome!</h1>
    <p>This is a paragraph with a <a href="index.html">link</a> in
it.</p>
    <ul>
      <li>first item</li>
      <li>second item</li>
      <li>third item</li>
    </ul>
  </body>
</html>
```

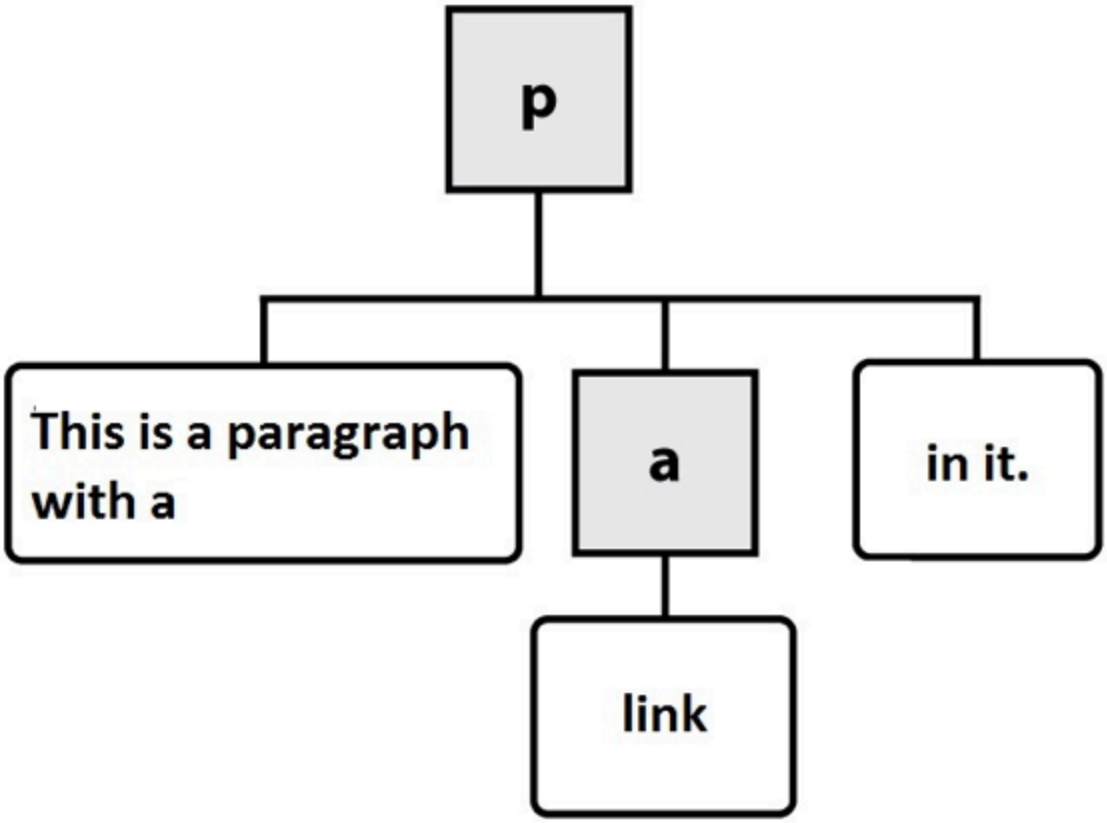
The browser will parse and render this into a tree of nodes, the DOM Tree:



The DOM Tree is made up of DOM Nodes, which represent all aspects of our document, from elements to attributes and comments. We'll refer to nodes and elements interchangeably, because all elements are nodes in the tree. However, there are also other types of nodes, for example: text nodes (the text in a block element) and attribute nodes (key/value pairs). We don't always show every node in our diagrams. Consider the `<p>` element from the example above:

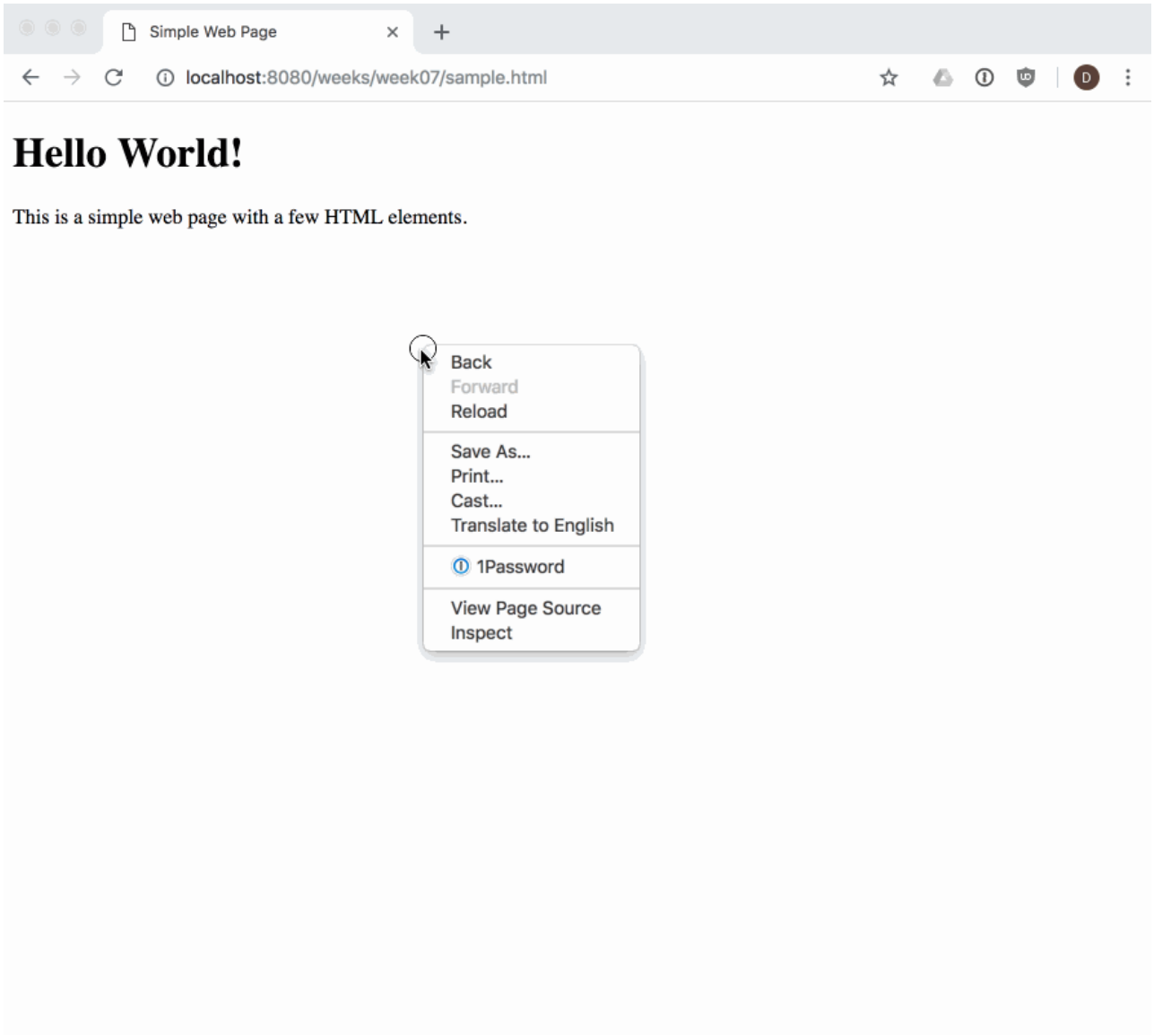
```
<p>This is a paragraph with a <a href="index.html">link</a> in
it.</p>
```

Here are the nodes that would be created:



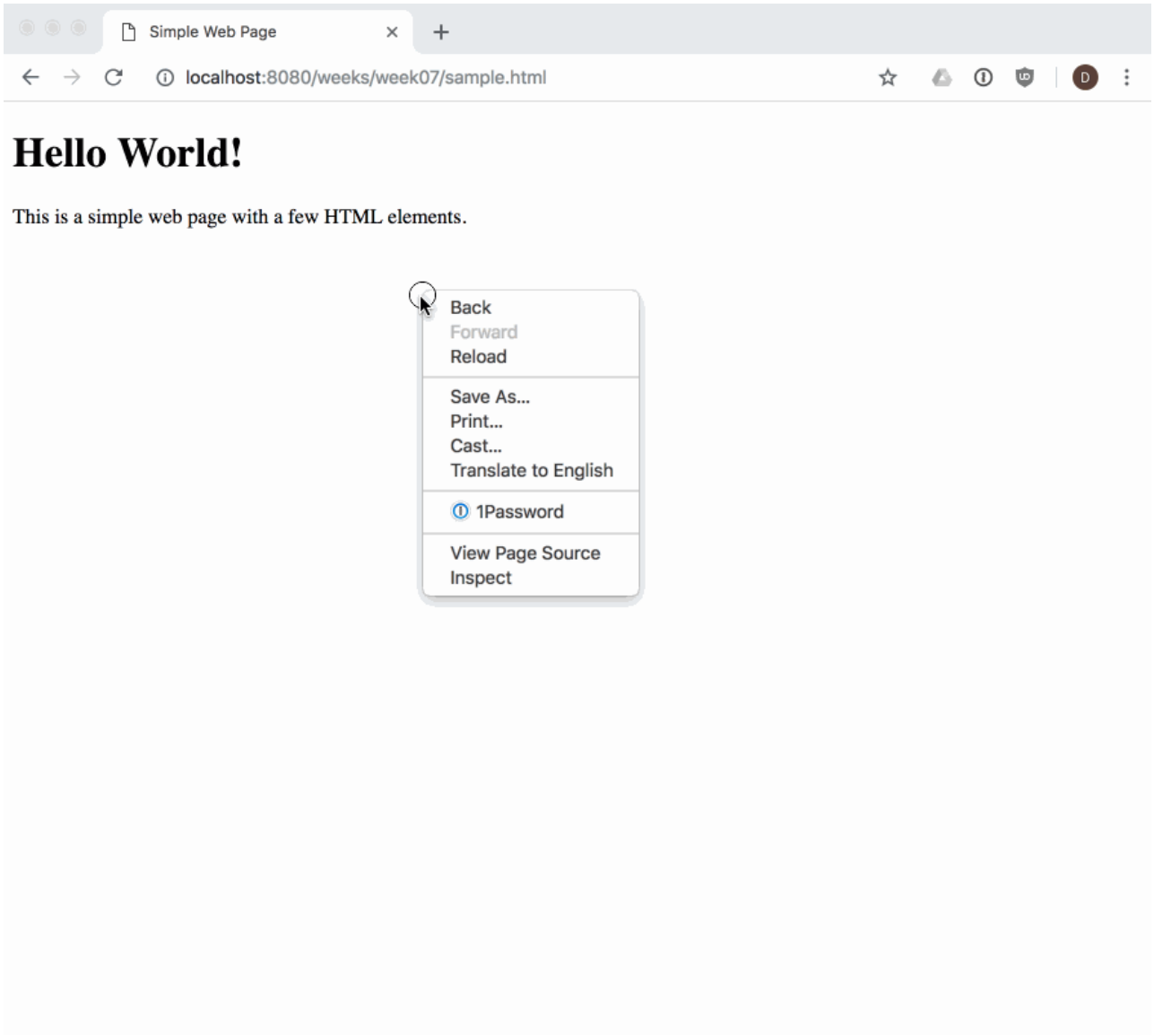
In this diagram, the gray, square boxes represent element nodes, while the white, rounded boxes are text nodes.

When we load a web page in a web browser, we see its fully parsed and rendered form. The web browser *begins* with the initial content we provide in our HTML. We can see the initial source HTML for any page we visit, whether we authored it or not:

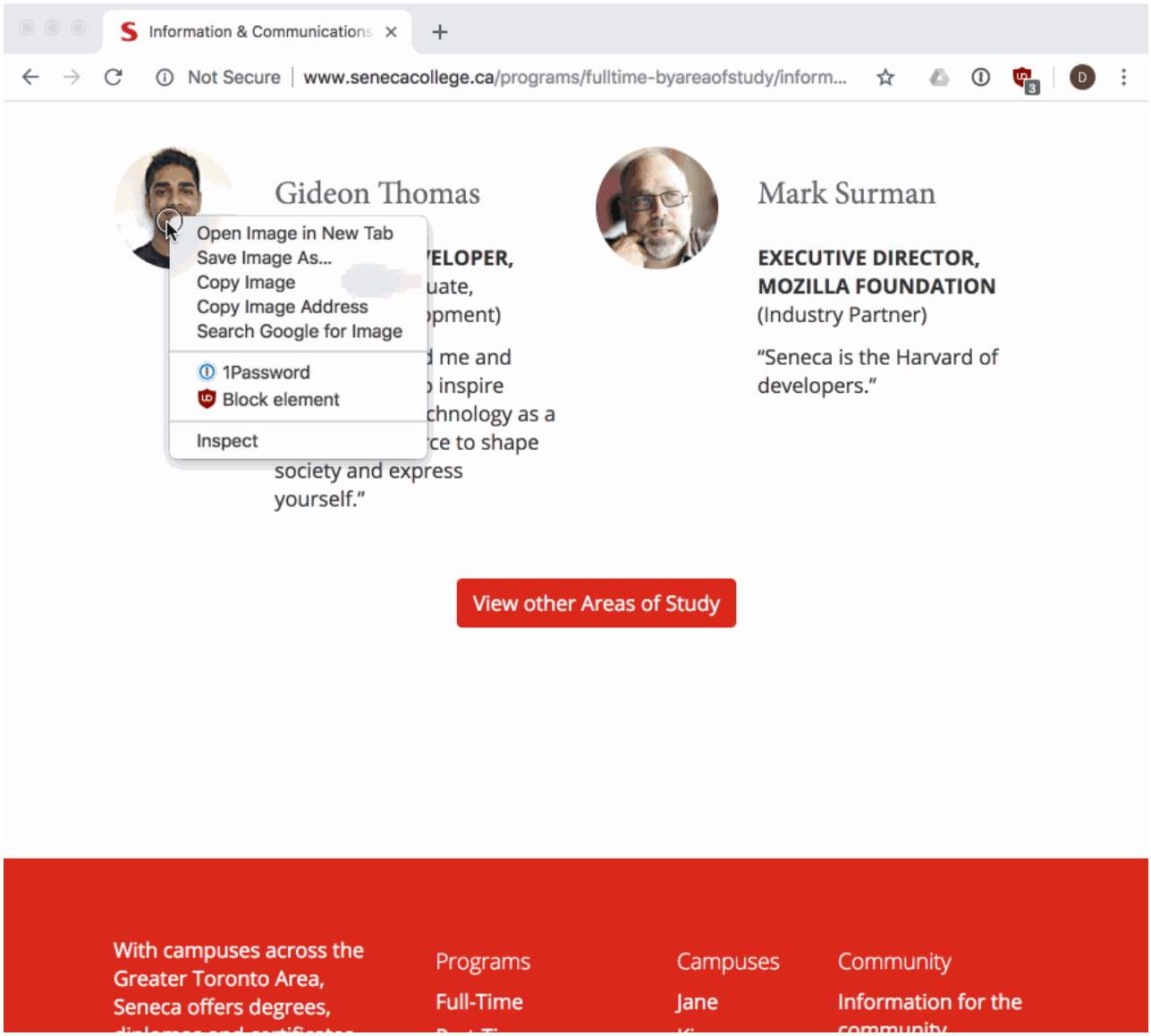


Our DOM Tree gets its name because of its shape: a *root* element connected to *child* nodes that extend like the branches of a tree. This tree structure is how the browser views our web page, and is why it is so important for us to open and close our HTML tags in order (i.e., our tags define the structure of the tree that the browser will create at runtime).

As web developers we can see and interact with the DOM tree for a page using the browser's built-in developer tools:



The dev tools allow us to view and work with the parsed DOM elements in a page. We can also use the dev tools to visually select an element in the page, and find its associated DOM element:



> NOTE: it's a good idea to get experience using, and learn about your browser's dev tools so that you can debug and understand when things go wrong while you are doing web development. There are a number of guides to help you learn, like this one from Google.

Programming the DOM

Web pages are dynamic: they can change in response to user actions, different

data, JavaScript code, etc. Where HTML defines the initial structure and content of a page, the DOM is the *current* or *actual* content of the page as it exists right now in your browser. And this can mean something quite different from the initial HTML used to load the page.

Consider a web page like GMail (or another email web client). When you visit your Inbox, the messages you see are not the same as when your friend visits hers. The HTML for GMail is the same no matter who loads the page. But it quickly changes in response to the needs of the current user.

So how does one modify a web page after it's been rendered in the browser? The answer is DOM programming. We've been using this "DOM" acronym without defining it, and its high time we did.

The Document Object Model (DOM) is a programming interface (i.e., set of Objects, functions, properties) allowing scripts to interact with, and modify documents (HTML, XML). The DOM is an object-oriented representation of a web page. Client-side web programming is essentially *using* the DOM via JavaScript to make web pages *do* things or *respond* to actions (e.g., user actions).

You may have noticed in our work with JavaScript that there was nothing particularly "webby" about it as a language: we wrote functions, worked with arrays, created objects. Lots of programming languages let you do this. JavaScript can't do anything with the web on its own. Instead, we need to access and use the Objects, functions, and properties made available to us by the DOM using JavaScript.

As web programmers we use the DOM via JavaScript to accomplish a number of important tasks:

1. Finding and getting references to elements in the page
2. Creating, adding, and removing elements from the DOM tree

3. Inspecting and modifying elements and their content
4. Run code in response to events triggered by the user, browser, or other parts of our code

Let's look at each one in turn.

Finding elements in the DOM with JavaScript

Our entry point to the DOM from JavaScript is via the global variable `window`. Every web page runs in an environment created by the browser, and that environment includes a global variable named `window`, which is provided by the browser (i.e., we don't create it).

There are hundreds of Objects, methods, and properties available to our JavaScript code via `window`. One example is `window.document`, which is how we access the DOM in our code:

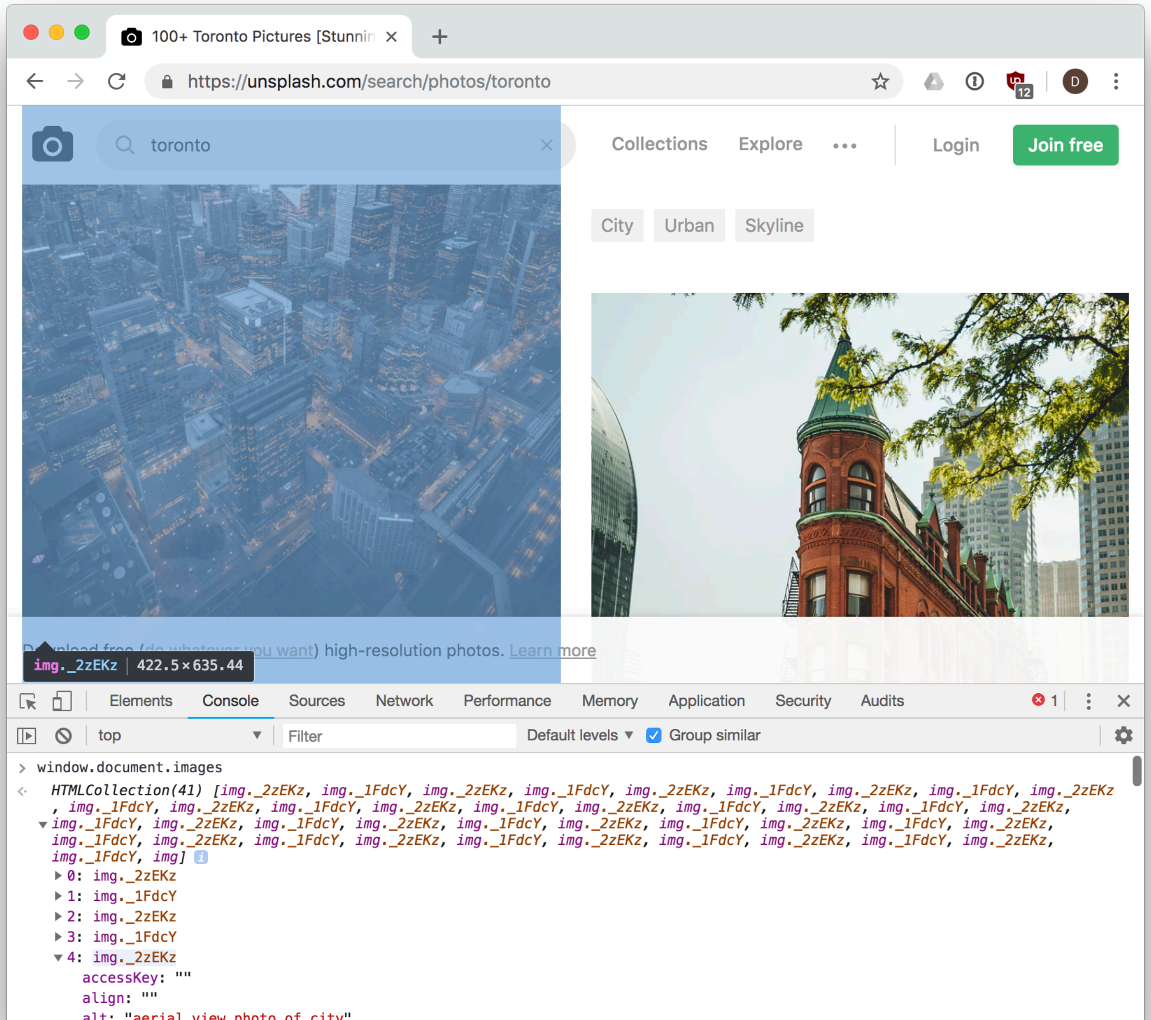
```
// Access the document object for our web page, which is in the
current window
let document = window.document;
```

> NOTE: since properties like `document` are available on the global `window` object, it is common to simply write `document` instead of `window.document`, since the global object is implied if no other scope is given.

Our document's tree of elements are now accessible to us, and we can access a number of well-known elements by name, for example:

```
// Get the value of the document's <title>
let title = document.title;
```

There are lots more. We can easily experiment with these in the dev tools web console, where we can access our `window` object. For example, here is the web page https://unsplash.com/search/photos/toronto with the web console open, and the result of `window.document.images` is shown, 41 `<img>` elements are returned in a collection:



We can also use a number of methods to search for and get a reference to one or more elements in our document:

- `document.getElementById(id)` - returns an element whose `id` attribute/ property has the given `id` `String`

```
<div id="menu">...</div>
<script>
  let menuDiv = document.getElementById('menu');
</script>
```

- `document.querySelector(selectors)` - similar to `document.getElementById(id)`, but also allows querying the DOM using CSS selectors for an element that doesn't have a unique id:

```
<div id="menu">
  <p class="formatted">...</p>
</div>
<script>
  // We can specify we want to query by ID using a leading #
  let menuDiv = document.querySelector('#menu');
  // We can specify we want to query by CLASS name using a
leading .
  let para = document.querySelector('.formatted');
</script>
```

- `document.querySelectorAll(selectors)` - similar to `document.querySelector(selector)`, but returns *all* elements that match the selectors as a `NodeList`:

```
<div id="menu">
  <p class="formatted">Paragraph 1...</p>
  <p class="formatted">Paragraph 2...</p>
  <p class="formatted">Paragraph 3...</p>
</div>
<script>
  // Get all <p> elements in the document as a list
  let pElements = document.querySelectorAll('p');
```

These four methods will work in any situation where you need to get a reference to something in document. In fact, you could rely solely on `document.querySelector()` and `document.querySelectorAll()`, which cover the same functionality as a number of other DOM methods:

```
// The following two lines of code do exactly the same thing.
// NOTE the use of # to indicate `demo` is an id in the second example.
let elem = document.getElementById('demo');
let elem = document.querySelector('#demo');
```

Creating elements and Modifying the DOM with JavaScript

In addition to searching through the DOM using JavaScript, we can also make changes to it. The DOM provides a number of methods that allow use to create new content:

- `document.createElement(name)` - creates and returns a new element of the type specified by `name`.

  ```
  let paragraphElement = document.createElement('p');
  let imageElement = document.createElement('img');
  ```

- `document.createTextNode(text)` - creates a text node (the text within an element vs. the element itself).

  ```
  let textNode = document.createTextNode('This is some text to show in an element');
  ```

These methods create the new nodes, but do not place them into the page. To do that, we first need to find the correct position within the existing DOM tree,

and then add our new node. We have to be clear *where* we want the new element to get placed in the DOM.

For example, if we want to place a node at the end of the `<body>`, we could use `.appendChild()`:

```
let paragraphElement = document.createElement('p');
document.body.appendChild(paragraphElement);
```

If we instead wanted to place it within an existing `<div id="content">`, we'd do this:

```
let paragraphElement = document.createElement('p');
let contentDiv = document.querySelector('#content');
contentDiv.appendChild(paragraphElement);
```

Both examples work the same way: given a parent node (`document` or `<div id="content">`), add (append to the end of the list of children) our new element.

We can also use `.insertBefore(new, old)` to accomplish something similar: add our new node before the `old` (existing) node in the DOM:

```
let paragraphElement = document.createElement('p');
let contentDiv = document.querySelector('#content');
let firstDivParagraph = contentDiv.querySelector('p');
contentDiv.insertBefore(paragraphElement, firstDivParagraph);
```

Removing a node is similar, and uses `removeChild()`:

```
// Remove a loading spinner
let loadingSpinner = document.querySelector('#loading-spinner');
// Get a reference to the loading spinner's parent element
let parent = loadingSpinner.parentNode;
parent.removeChild(loadingSpinner);
```

Examples

1. Add a new heading to a document

```
// Create a new <h2> element
let newHeading = document.createElement('h2');

// Add some text to the <h2> element we just created.
// Similar to doing <h2>This is a heading</h2>.
let textNode = document.createTextNode('This is a heading');
// Add the textNode to the heading's child list
newHeading.appendChild(textNode);

// Insert our heading into the document, at the end of <body>
document.body.appendChild(newHeading);
```

2. Create a new paragraph and insert into the document

```
<div id="demo"></div>
<script>
  // Create a <p> element
  let pElem = document.createElement('p');

  // Use .innerHTML to create text nodes inside our <p>...</p>
  pElem.innerHTML = 'This is a paragraph.';

  // Get a reference to our <div> with id = demo
```

Inspecting, Modifying a DOM element with JavaScript

Once we have a reference to an element in JavaScript, we use a number of properties and methods to work with it.

Element Properties

- `element.id` - the `id` of the element. For example: `<p id="intro"></p>` has an `id` of `"intro"`.
- `element.innerHTML` - gets or sets the markup contained within the element, which could be text, but could also include other HTML tags.
- `element.parentNode` - gets a reference to the parent `node` of this element in the DOM.
- `element.nextSibling` - gets a reference to the sibling element of this element, if any.
- `element.className` - gets or sets the value of the `class` attribute for the element.

Element Methods

- `element.querySelector()` - same as `document.querySelector()`, but begins searching from `element` vs. `document`
- `element.querySelectorAll()` - same as `document.querySelectorAll()`, but begins searching from `element` vs. `document`
- `element.scrollIntoView()` - scrolls the page until the element is in view.
- `element.hasAttribute(name)` - checks if the attribute `name` exists on this

`element`

- `element.getAttribute(name)` - gets the value of the attribute `name` on this `element`
- `element.setAttribute(name, value)` - sets the `value` of the attribute `name` on this `element`
- `element.removeAttribute(name)` - removes the attribute `name` from this `element`

Examples

1. Reveal an error message in the page, by removing an element's `hidden` attribute

```
<!-- The `hidden` attribute means this <div> won't be displayed
until it's removed -->
<div id="error-message" hidden>
  <p>There was an error saving the document. Please try
again!</p>
</div>
<script>
  // Try to save the file, and
  let error = saveFile();
  if (error) {
    let elem = document.querySelector('#error-message');
    elem.removeAttribute('hidden');
  }
</script>
```

2. Insert a user's profile picture into the page

```
// Insert the user's picture (e.g., in response to hovering
over a username)
let profilePic = document.createElement('img');

// Set attributes via getters/setters on the element vs.
attributes
profilePic.id = 'user-' + username;
profilePic.height = 50;
profilePic.src = './images/' + username + '-user-profile.jpg';

// Insert the profile pic <img> into the document
document.body.appendChild(profilePic);

// Make sure the new image is visible, or scroll until it is
profilePic.scrollIntoView();
```

3. Add new paragraph elements to a div

```
// Use .innerHTML as a getter and setter to update some text
let elem = document.querySelector('#text');

elem.innerHTML = '<p>This is a paragraph</p>';
elem.innerHTML = elem.innerHTML + '<p>This is another
paragraph</p>';
```

Events & Timers

Events

The DOM relies heavily on a concept known as event-driven programming. In event-driven programs, a main loop (aka the *event loop*), processes events as they occur.

Examples of events include things like user actions (clicking a button, moving the mouse, pressing a key, changing tabs in the browser), or browser/code initiated actions (timers, messages from background processes, reports from sensors).

Instead of writing a program in a strict order, we write functions that should be called in response to various events occurring. Such functions are often referred to as *event handlers*, because they handle the case of some event happening. If there is no event handler for a given event, when it occurs the browser will simply ignore it. However, if one or more event handlers are registered to listen for this event, the browser will call each event handler's function in turn.

> You can think of events like light switches, and event handlers like light fixtures: flipping a light switch on or off triggers an action in the light fixture, or possibly in multiple light fixtures at once. The lights handle the event of the light switch being flipped.

DOM programming is typically done by writing many functions that execute in response to events in the browser. We register our event handlers to indicate that we want a particular action to occur. DOM events have a `name` we use to refer to them in code.

We can register a DOM event handler for a given event in one of two ways:

1. `element.onevent = function(e) {...};`
2. `element.addEventListener('event', function(e) {...})` and `element.removeEventListener('event', function(e) {...})`

In both cases above, we first need an HTML element. Events are emitted to a *target* element. Elements in the DOM can trigger one or more events, and we must know the name of the event we want to handle.

In the first method above, `element.onevent = function(e) {...};`, a single event handler is registered for the `event` event connected with the target element `element`. For example, `document.body.onclick = function(e) {...};`, indicates we want to register an event handler for the `click` event on the `document.body` element (i.e., `<body>...</body>`).

In the second method above, use `addEventListener()` to add as many individual, separate event handlers as we need. Whereas `element.onclick = function(e) {...};` binds a single event handler (function) to the `click` event for `element`, using `element.addEventListener('click', function(e) {...});` *adds* a new event handler (function) to any that might already exist.

Consider the following code:

```
let body = document.body;

function handleClick(e) {
  // Process the click event
}

function handleClick2(e) {
  // Another click handler
}
```

Because `addEventListener()` is more versatile than the older `onevent` properties, you are encouraged to use it in most cases.

Here's an example of the first method, where we only need a single event handler. In the following case, a web page has a Save button, and we want to save the user's work when she clicks it.

```
<button id="btn-save">Save</button>
<script>
  // Get a reference to our Save <button>
  let saveBtn = document.querySelector('#btn-save');

  function save() {
    // Save the user's work
  }

  // Register a single event handler on the save button's click
event
  saveBtn.onclick = function (e) {
    // Save the user's work, calling a save() function we wrote
elsewhere
    save();
  };
</script>
```

Now consider the same code, but with multiple event handlers. In this case we not only want to save the user's work, but also log the information in our web analytics so we can keep track of how popular this feature is (how many times it gets clicked):

```
<span id="needs-saving">Document has changes, Remember to
Save!</span>
...
```

In this second example, it's possible for the browser to call more than one function (event handler) in response to a single event (`click`). What's nice about this is that different parts of our code don't have to be combined into a single function. Instead, we can keep things separate (saving logic vs. analytics logic).

A complete example of a page that listens for changes to the network online/offline status, and updates the page accordingly, is available at online.html.

Common Events

There are many types of events we can listen for in the DOM, some of which are very specialized to certain elements or Objects. However, there some common ones we'll use quite often:

- `load` - fired when a resource has finished loading (e.g., a `window`, `img`)
- `beforeunload` - fired just before the window is about to be unloaded (closed)
- `focus` - when the element receives focus (cursor input)
- `blur` - when the element loses focus
- `click` - when the user single clicks on an element
- `dblclick` - when the user double clicks on an element
- `contextmenu` - when the right mouse button is clicked
- `keypress` - when a key is pressed on the keyboard
- `change` - when the content of an element changes (e.g., an input element in a form)
- `mouseout` - when the user moves the mouse outside the element
- `mouseover` - when the user moves the mouse over top of the element
- `resize` - when the element is resized

All of the events described above can be used in either of the two ways we discussed above. For example, if we wanted to use the `mouseout` event on an element:

```
<div id="map">...</div>
<script>
  let map = document.querySelector('map');

  // Method 1: register a single event handler via the on* property
  map.onmouseout = function (e) {
    // do something here in response to the mouseout event on this div.
  };

  // Method 2: register one of perhaps many event handlers via addEventListener
  map.addEventListener('mouseout', function (e) {
    // do something here in response to the mouseout event on this div.
  });
</script>
```

The `Event` Object

In the example code above, you may have noticed that our event handler functions often looked like this:

```
element.onclick = function (e) {
  // e is an instance of the Event object
};
```

The single `e` argument is an instance of the `Event` Object. The `e` or `event` is

provided to our event handler function in order to pass information about the event, and to give us a chance to alter what happens next.

For example, we can get a reference to the element to which the event was dispatched using `e.target`. We can also instruct the browser to prevent the "default" action from happening as a result of this event using `e.preventDefault()`, or stop the event from continuing to *bubble* up the DOM (i.e., rise up the DOM tree nodes, triggering other event handlers along the way) using `e.stopPropagation().

Here's an example showing how to use these:

```
<button id="btn">Click Me</button>
<script>
  document.querySelector('#btn').addEventListener('click',
function (e) {
    // Prevent this event from doing anything more, we'll handle
it all here.
    e.preventDefault();
    e.stopPropagation();

    // Get a reference to the <button> element
    let btn = e.target;

    // Change the text of the button
    btn.innerHTML = 'You clicked Me!';
  });
</script>
```

Some events also provide specialized (i.e., derived from `Event`) `event` Objects with extra data on them related to the context of the event. For example, a `MouseEvent` gives extra detail whenever a click, mouse move, etc. event occurs:

```
<div id="position"></div>
<script>
  document.body.addEventListener('click', function (e) {
    // Get extra info about this mouse event so we know where the
pointer was
    let x = e.screenX;
    let y = e.screenY;

    // Display co-ordinates where the mouse was clicked: "Position
(300, 342)"
    document.querySelector('#position').innerHTML = `Position
(${x}, ${y})`;
  });
</script>
```

Timers

It's also possible for us to write an event handler that happens in response to a timing event (delay) vs. a user or browser event. Using these timing event handlers, we scheduling a task (function) to run after a certain period of time has elapsed:

- `setTimeout(function, delayMS)` - schedule a task (`function`) to be run in the future (`delayMS` milliseconds from now). Can be cancelled with `clearTimeout(timerID)`
- `setInterval(function, delayMS)` - schedule a task (`function`) to be run in the future every `delayMS` milliseconds from now. Function will be called repeatedly. Can be cancelled with `clearInterval(timerID)`

Here's an example of using an `interval` to update a web page with the current date and time every 1 second.

```
<p hidden>The current date and time is <time id="current-
date"></time></p>
<button id="btn-start">Start Timer</button>
<button id="btn-end">End Timer</button>
<script>
  let startButton = document.querySelector('#btn-start');
  let endButton = document.querySelector('#btn-end');
  let timerId;

  // When the user clicks Start, start our timer
  startButton.onclick = function (e) {
    // If the user clicks it more than once, ignore it once it's
running
    if (timerId) {
      return;
    }

    let currentDate = document.querySelector('#current-date');
    currentDate.removeAttribute('hidden');

    // Start our timer to update every 1000ms (1s), showing the
current date/time.
    timerId = setInterval(function () {
      let now = new Date();
      currentDate.innerHTML = now.toLocaleString();
    }, 1000);
  };

  endButton.onclick = function (e) {
    // If the user clicks End when the timer isn't running, ignore
it.
    if (!timerId) {
      return;
    }

    // Stop the timer
    clearInterval(timerId);
```

Practice Exercise

In this exercise, we will practice working with HTML, images, URLs, the DOM, events, and JavaScript to create an interactive web page.

1. Create a **folder** called `cats` on your computer
2. Create a **file** inside the `cats` folder named `index.html`
3. Open a **terminal** to your `cats` folder (i.e., `cd cats`)
4. In your **terminal**, start a web server by running the following command: `npx http-server` (alternatively, you can use the command: `npx lite-server`, refer week 5 notes)
5. Open the `cats` folder in Visual Studio Code
6. Edit the `index.html` file so it contains a basic HTML5 web page, including a `<head>`, `<body>`, etc. Try to do it from memory first, then look up what you've missed.
7. Save `index.html` and try loading it in your browser by visiting your local web server at `http://localhost:8080/index.html`
8. In your editor, modify the `body` of your `index.html` file to contain the text of the poem in cats.txt. Use HTML tags to markup the poem for the web. Your page should have a proper heading for the title, each line should break at the correct position, and the poet's name should be bold.
9. Add an image of a cat to the page below the text. You can use https://upload.wikimedia.org/wikipedia/commons/c/c1/Six*weeks_old_cat*%28aka%29.jpg.

10. Adjust the `width` of your image so it fits nicely on your page. What happens if you adjust the `width` and `height`?

11. Create a new file in your `cats` folder called `script.js`. Add the following line of JavaScript:

```
console.log('cats!');
```

12. Add a `script` element to the bottom of your `body` (i.e., right before the closing `</body>` tag). Set its `src` to a file called `script.js`:

```
    <script src="script.js"></script>
</body>
```

13. Refresh your web page in the browser, and open your browser's `Dev Tools`, and `Web Console`. Make sure you can see the `cats!` message printed in the log.

14. Try changing `cats!` in `script.js` to some other message, save your `script.js` file, and refresh your browser. Make sure your console updates with the new message.

15. Modify `index.html` and update your `<img>` tag: add an attribute `id="cat-picture"` and remove the `src="..."`:

```
<!-- NOTE: there is no longer a src attribute in our HTML,
we'll do it JavaScript below -->
<img id="cat-picture" />
```

16. Modify your `script.js` file to add the following code:

```
window.onload = function () {
  let img = document.getElementById('cat-picture');
  img.src =
    'https://upload.wikimedia.org/wikipedia/commons/c/c1/
Six_weeks_old_cat_%28aka%29.jpg';
};
```

17. Save your `script.js` file and reload your browser. Do you still see a cat? If not, check your web console for any errors.

18. Modify your `script.js` and change your cat URL used by `img.src` to use https://cataas.com/cat. The cataas.com site provides cat pictures as a service via URL parameters. Save `script.js` and reload your page a few times. Do you see a different cat each time?

19. Modify your `script.js` file to move your image code to a separate function. Make sure it still works the same way when you're done (save and test in your browser):

```
function loadCatPicture() {
  let img = document.getElementById('cat-picture');
  img.src = 'https://cataas.com/cat';
}

window.onload = loadCatPicture;
```

20. Rewrite `script.js` to update the picture after 5 seconds:

```
function loadCatPicture() {
  let img = document.getElementById('cat-picture');
  img.src = 'https://cataas.com/cat';
}
```

21. Rewrite `script.js` to update the picture every 15 seconds, forever:

```
function loadCatPicture() {
  let img = document.getElementById('cat-picture');
  img.src = 'https://cataas.com/cat';
}

window.onload = function () {
  loadCatPicture();

  // Call the loadCatPicture function every 15000ms
  setInterval(loadCatPicture, 15 * 1_000 /* 15s = 15000ms */);
};
```

22. Rewrite `script.js` to update the picture only when the user clicks somewhere in the window:

```
function loadCatPicture() {
  let img = document.getElementById('cat-picture');
  img.src = 'https://cataas.com/cat';
}

window.onload = function () {
  loadCatPicture();

  // Call the loadCatPicture function when the user clicks in
the window
  window.onclick = loadCatPicture;
};
```

23. Modify `index.html` and put a `<div>...</div>` around all the text of the poem. Give your `div` an `id="poem-text"` attribute:

```
<div id="poem-text">
  <p>Cats sleep anywhere, any table, any chair....</p>
  ...
</div>
```

24. Rewrite `script.js` to load the picture only when the user clicks on the text of the poem:

```
function loadCatPicture() {
  let img = document.getElementById('cat-picture');
  img.src = 'https://cataas.com/cat';
}

let poemText = document.getElementById('poem-text');
poemText.onclick = loadCatPicture;
```

25. Rewrite `script.js` to also load the picture only when the user presses a key on the keyboard:

```
function loadCatPicture() {
  let img = document.getElementById('cat-picture');
  img.src = 'https://cataas.com/cat';
}

let poemText = document.getElementById('poem-text');
poemText.onclick = loadCatPicture;

window.onkeypress = function (event) {
  let keyName = event.key;
  console.log('Key Press event', keyName);
  loadCatPicture();
};
```

26. Rewrite `script.js` to also load the picture only when the user presses a key on the keyboard, but only one of `b, m, s, n, p, x`:

```
function loadCatPicture() {
  let img = document.getElementById('cat-picture');
  img.src = 'https://cataas.com/cat';
}

let poemText = document.getElementById('poem-text');
poemText.onclick = loadCatPicture;

window.onkeypress = function (event) {
  let keyName = event.key;
  console.log('Key Press event', keyName);

  switch (keyName) {
    case 'b':
    case 'm':
    case 's':
    case 'n':
    case 'p':
    case 'x':
      loadCatPicture();
      break;
    default:
      console.log('Ignoring key press event');
  }
};
```

27. Rewrite `script.js` to also load the picture only when the user presses a key on the keyboard, but only one of `b, m, s, n, p, x`, and load the picture with one of the supported cataas filters:

```
function loadCatPicture(filter) {
  let url = 'https://cataas.com/cat';
  let img = document.getElementById('cat-picture');

  // If the function is called with a filter argument, add that
to URL
  if (filter) {
    console.log('Using cat picture filter', filter);
    url += `?filter=${filter}`;
  }

  img.src = url;
}

let poemText = document.getElementById('poem-text');
poemText.onclick = function () {
  loadCatPicture();
};

window.onkeypress = function (event) {
  let keyName = event.key;
  console.log('Key Press event', keyName);

  switch (keyName) {
    case 'b':
      return loadCatPicture('blur');
    case 'm':
      return loadCatPicture('mono');
    case 's':
      return loadCatPicture('sepia');
    case 'n':
      return loadCatPicture('negative');
    case 'p':
      return loadCatPicture('paint');
    case 'x':
      return loadCatPicture('pixel');
```

28. Rewrite `script.js` so that we only load a new cat picture when the old picture is finished loading (don't send too many requests to the server). Also, add some cache busting:

```
// Demonstrate using a closure, and use an immediately
executing function to hide
// an `isLoading` variable (i.e., not global), which will keep
track of whether
// or not an image is being loaded, so we can ignore repeated
requests.
let loadCatPicture = (function () {
  let isLoading = false;

  // This is the function that will be bound to loadCatPicture
in the end.
  return function (filter) {
    if (isLoading) {
      console.log('Skipping load, already in progress');
      return;
    }

    let img = document.getElementById('cat-picture');

    function finishedLoading() {
      isLoading = false;

      // Remove unneeded event handlers so `img` can be garbage
collected.
      img.onload = null;
      img.onerror = null;
      img = null;
    }
    img.onload = finishedLoading;
    img.onerror = finishedLoading;
```

Introduction to CSS & Syntax

In HTML5 we don't include markup related to how our page should look; instead we focus on its structure, layout, and organization. We put all this information in style sheets: text files that define CSS *selectors* and *rules* for how to style our HTML elements.

CSS allows us to specify styles, layout, positioning, and other "style" properties for HTML elements. CSS makes it possible for a page's style information to be separated from its structure and content. Consider how much of an impact CSS can have on the same HTML:

- CSS Zen Garden
- CSS Zen Garden HTML file
- CSS Zen Garden CSS file

CSS Syntax

CSS syntax is made up of *rules*, which are broken into two parts:

1. a *selector*, specifying the element(s) that should have the rules applied
2. one or more *declarations*, which are *key/value* pairs surrounded by `{...}` braces

```
h1 {
  color: blue;
  font-size: 12px;
```

In this example, the *selector* is `h1`, which indicates that we want the following rules to be applied to level-1 heading elements (i.e., all `<h1></h1>` elements in the document). Next comes a list of two definitions, each ending with a `;`. These declarations follow the usual key/value syntax, with a *property* name coming before the `:`, and a *value* coming after:

- `color: blue;` says we want to use the colour (note the spelling) blue
- `font-size: 12px;` says we want the font to be 12px.

Here's another example:

```
p {
  color: red;
  text-align: center;
  text-decoration: underline;
}
```

This indicates we want all `<p></p>` elements in the document to have red, centered, underlined text.

Where to Put CSS

CSS can come from a number of sources in an HTML page:

1. Inline
2. Internal Embedded
3. External File(s)
4. The browser itself (e.g., default styles, or extra styles injected by a browser extension)

Browsers apply styles to elements using a priority order that matches the list

above. If more than one style rule is specified for an element, the browser will prefer whatever is defined in Inline styles over Internal Embedded, Internal Embedded over External files, etc.

Inline Example

CSS rules can be placed directly on an element via the `style` attribute:

```
<div style="background-color: green">...</div>
```

Internal Embedded

If we want to apply the same CSS rules to more than one element, it makes more sense to *not* duplicate them on every element's `style` attribute. One solution is to use an internal embedded `<style>` element in the `<head>` or `<body>`, similar to how embedded `<script>` elements work:

```
<style>
  p {
    color: red;
  }

  div {
    background-color: blue;
    text-align: center;
  }
</style>
```

External File(s)

Putting large amounts of CSS in `<style>` elements makes our HTML harder to

read and maintain (CSS is about separating style from structure), and also causes our page to perform worse in terms of load times (i.e., the styles can't be cached by the browser). To overcome this, we often include external `.css` files via the `<link>` element within the document's `<head>`:

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="styles.css" type="text/css" />
  </head>
</html>
```

We can include many stylesheets in this way (i.e., everything doesn't have to go in one file), and we can include `.css` files on the same origin, or a remote origin:

```
<!doctype html>
<html>
  <head>
    <link
      rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/
css/bootstrap.min.css"
    />
    <link rel="stylesheet" href="styles.css" type="text/css" />
  </head>
</html>
```

In the example above, the page uses the popular Bootstrap CSS styles along with some locally (i.e., local to the web server) styles in `styles.css`.

A `.css` file included in this way can also `@import` to have even more `.css` files get loaded at runtime:

```
/* Import Font Awesome */
@import url(https://use.fontawesome.com/releases/v5.4.2/css/
all.css);
```

In this example, the popular Font Awesome CSS library for font icons has been imported via a `.css` file.

CSS Selectors

We've already learned a few CSS Selectors when we discussed `querySelector()` and `querySelectorAll()`. The word `Selector` refers to the fact that these methods take a CSS Selector and return DOM elements that match. For example:

- `document.querySelector('#output')` would return the element with attribute `id="output"`
- `document.querySelectorAll('.logo')` would return all elements with a class of `logo`
- `document.querySelectorAll('img')` would return all `<img>` elements

These same selectors, and many more, can also be used in our CSS rulesets.

Tag/Type Selectors

The name of an HTML element can be used to specify the styles associated with all elements of the given type. For example, to indent all `<p>` text in our document, we could do this:

```
p {
```

Class Selectors

Often we want to apply styles to *some* but not *all* elements of a certain kind. Perhaps we only want some of our page's `<p>` elements to have a particular look. To achieve this, we define a *class*, and then put that class on the elements that require it:

```
<style>
  .demo {
    text-decoration: underline red;
  }
</style>

<p>This is a paragraph that won't get the styles below applied to
it (doesn't include the class)</p>
<p class="demo">This paragraph will get the styling applied.</p>
<p class="demo">And so will this one.</p>
```

A class can be applied to elements that aren't of the same type:

```
<style>
    .invisible {
        display: none;
    }
</style>

<h1 class="invisible">Title</p>
<p class="invisible">This is a paragraph.</p>
```

I this example, both the `<h1>` element, and the `<p>` element will have the `display: none` style applied, hiding them so they don't appear in the page.

If we want to be more specific, and only apply styles to elements of a given type which also have a given class, we can do this:

```
<style>
  p.note {
    font-weight: bold;
  }
</style>

<p class="note">This is a paragraph that also uses the note
class.</p>
<div class="note">
  This div uses the note class too, but because we said p.note, no
styles are used.
</div>
```

An element can also have multiple classes applied, each one adding different styling:

```
<style>
  .invisible {
    display: none;
  }

  .example {
    color: green;
    background-color: red;
  }
</style>

<p class="invisible example">This is a paragraph that uses two
classes at once.</p>
```

ID Selectors

In many cases, we have only a single element that should use styles. Using a type or class selector would be overly broad, and so we tend to use an `id` instead. Recall that only one HTML element in a document can have a given `id` attribute: it must be unique.

```
<style>
  #summary {
    background-color: skyblue;
  }
</style>

<div id="summary"></div>
```

When we use the `id` as a selector, we prefix it with the `#` symbol. Notice that the HTML does *not* use the `#` symbol though.

Contextual Selectors

Another common way to write selectors is to use the position of elements in the DOM. The *context selector* indicates the context, or placement/nesting (i.e., determined by the parent node) of the element.

For example, if we want to apply styles to `<p>` elements that are children of `<div>` elements, we could do this:

```
<style>
  div p {
    font-size: 16px;
  }
```

Grouping Selectors

As our CSS grows, it's common that we'll notice that we're repeating the same things multiple times. Instead of doing this, we can group a number of selectors together into a comma-separated list:

```
html,
body {
  height: 100%;
}

h1,
h2,
h3,
h4,
h5,
h6 {
  font-family: Serif;
  color: blue;
}
```

Here we've used grouping twice to cut-down on the number of times we have to repeat things. In the first case, we defined a height of `100%` (full height of the window) for the `<html>` and `<body>` elements (they don't have a height by default, and will only be as tall as the content within them). We've also declared some font and color information for all the headings we want to use.

Suggested Readings

- Introduction to CSS
- Learning to Style HTML using CSS
- CSS: Cascading Style Sheets on MDN

Applied CSS

Containers for Styling

We've discussed `<div>` and `<span>` in the past, but their purpose may not have been clear. Why bother wrapping other elements in `<div>...</div>` or `<span>...</span>` when they don't display any different?

With CSS we can now start to take advantage of what they provide. If we think of them as containers which can be used to group styling, their purpose will become more clear.

A `<div>` is a block level element, and `<span>` an inline element. Depending on how we want to group and apply styling, we can use one or both. Consider the following:

```
<style>
  .info-box {
    border: solid green;
  }

  .info-box p {
    font-family: Serif;
  }

  .info-box span {
    font-weight: bold;
  }

  .info-box img {
    width: 75px;
    height: 75px;
```

CSS Units

Many CSS values require units to be specified, for example, font sizes, widths, heights, etc. At first you might think that we should specify things in pixels; however, browsers need to work on such a wide variety of hardware and render to so many different displays (watches to billboards), we need more options. It's also important to be able to specify sizes using relative units vs. fixed, for layouts that need to adapt to changing conditions and still retain the correct proportions.

There is one exception, and that is for `0` (i.e., zero), which never needs a unit (i.e., `0px` is the same as `0%`, etc).

The most common units we use in CSS are:

```
1em = 12pt = 16px = 100%
```

Let's look at each of these in turn:

- `em` (the width of the capital letter `M`) - a scalable unit that is used in web media, and is equal to the current `font-size`. If the `font-size` is `12pt`, `1em` is the same as `12pt`. If the `font-size` is changed, `1em` changes to match. We can also use multiples: `2em` is twice the `font-size`, and `.5em` is half. Using `em` for sizes is popular on the web, since things have to scale on mobile vs. desktop (i.e., fixed unit sizes don't work as the screen shrinks/expands).
- `pt` - a fixed-size *Point* unit that comes from print media, where `1pt` equals `1/72` of an inch.
- `px` - pixels are fixed size units for web media (screens), and `1px` is equal

to one dot on a computer display. We use `px` on the web when we need "pixel perfect" sizing (e.g., image sizes).

- `%` - the percent unit is similar to `em` in that it scales with the size of the display. `100%` is the same as the current `font-size`.
- `vw`, `vh` - the viewport width and height units are percentages of the visible space in the viewport (the part of the page you can see, the window's width and height). `1vw` is the same as `1%` of the width of the viewport, and `80vh` is the same as `80%` of the visible height.

You will also sometimes encounter other ways of measurement that use full words: `xx-small, x-small, small, medium, large, x-large, xx-large, smaller, larger, thin, medium, thick`

Here's an example that uses a number of the units mentioned above:

```
<style>
  html,
  body {
    height: 100vh;
  }

  .box {
    margin: 10px;
    font-size: 2em;
    height: 150px;
    border: medium solid black;
  }
</style>
<div class="box"></div>
```

CSS Colours (`color`)

CSS allows us to define colour values for many declarations. We do so by specifying a colour using one of the following notations:

- Hexadecimal Red, Green, Blue: written using 3 double-digit hex numbers, and starting with a `#` sign. Each of the 3 pairs represents a value between 0 and 255 for Red, Green, and Blue: `#000000` is pure Black and `#ffffff` is pure White, and `#ffd700` is Gold.
- RGB or RGBA notation: here the red, green, blue, and sometimes alpha (i.e., opacity) are defined in decimal notation: `#ffffff` is the same as `rgb(255, 255, 255)` and `#ffd700` is the same as `rgb(255, 215, 0)`. If we want to define how see-through the colour is (by default you can't see through a colour), we add an alpha value: `rgba(0, 191, 0, 0.5)` means that the colour will be 50% see through.
- Named colours: some colours are so common that they have their own name defined in the CSS standard. For example: `white`, `black`, `green`, `red`, but also `chocolate`, `darkorange`, `peru`, etc.

The easiest way to understand this is using a Colour Picker tool, which lets you visually see the difference in changing values.

CSS Properties and Values

A *property* is assigned to a selector in order to manipulate its style. The CSS properties are defined as part of the CSS standard. When you want to know how one of them works, or which values you can assign, you can look at the documentation on MDN. For example:

- `text-indent`
- `color`
- `background-color`
- `border`

There are hundreds of properties we can tweak as web developers, and it's a good idea to explore what's available, and to look at how other web sites use them via the developer tools.

A property can have one or more values. A the possible values a property can have also comes from the standard. For example:

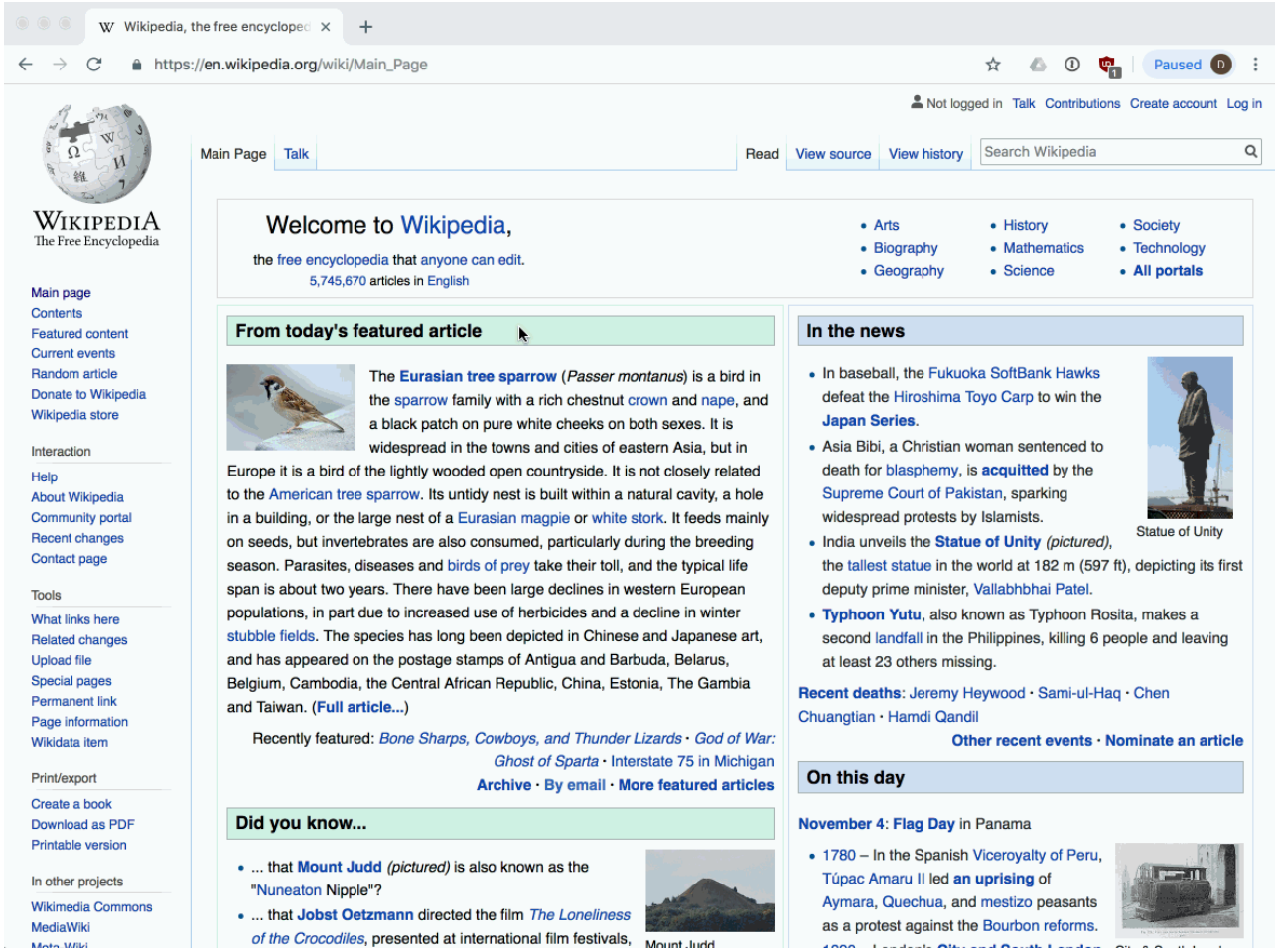
```
p {
  text-decoration: underline;
}

.spelling-error {
  text-decoration: red wavy underline;
}
```

The `text-decoration` property is defined to take one of a number of values, each of which is also defined in the standard.

Exploring CSS Properties and Values in the Dev Tools

By far the best way to learn about CSS is to look at how other sites use it. When you find something on the web that you think looks interesting, open your browser's dev tools and inspect the CSS Styles:



You can look at the specific properties specified for an element, or see all the *computed* styles (i.e., everything, including all default values). You can also try toggling these on and off, or double-click the values to enter your own.

CSS `text` Properties

There are dozens of properties that affect how text is rendered. These include things like the color, spacing, margins, font characteristics, etc.

```
h2 {
  color: red;
```

`font` Properties

We can use the `font-family` property to specify a font, or list of fonts, for the browser to apply to an element. The font must be available on the user's computer, otherwise the next font in the list will be tried until one is found that is installed, or a default font will be used.

In general it is safe to assume that the following fonts are available:

- `Helvetica, Arial, Verdana, sans-serif` - sans-serif fonts
- `"Courier New", Courier, monospace` - monospace fonts
- `Georgia, "Times New Roman", Times, serif` - serif fonts

You can see a list of the fonts, and OS support here.

```
h3 {
  font-family: Arial;
}

h4 {
  font-family: 'Times New Roman', Times, serif;
}

h5 {
  font-size: 18pt;
  font-style: italic;
  font-weight: 500;
}
```

Web Fonts - `@font-face`

Modern browsers also allow custom fonts to be included as external files, and

downloaded as needed by the web site. This is often the preferred method for designers, who don't want to be limited to the set of fonts available on *all* operating systems.

A font is a file that describes the curves and lines needed to generate characters at different scales. There are various formats, from `OTF` (OpenType Format) to `TTF` (TrueType Format) to `WOFF` (Web Open Font Format), etc. In order for the browser to use a new font, it has to be downloadable via one or more URLs. We then tell the browser which font files to download in our CSS via the `@font-face` property:

```
@font-face {
    font-family: "FontName"
    src: url(font.woff2) format('woff2'),
         url(font.ttf) format('truetype');
}

body {
    font-family: "FontName";
}
```

Many fonts have to be purchased, but there are some good sources of high quality, freely available fonts for your sites:

- Font Squirrel
- dafont.com
- Google Fonts

For example, we can use the popular `"Lobster"` font from Google by doing the following in our CSS:

```
@import url(https://fonts.googleapis.com/css?family=Lobster) p {
```

`font-size` property

Using the `font-size` property, font sizes can be given in fixed or relative units, depending on how we want our text to scale on different devices:

```
h1 {
  font-size: 250%; /* scaled to 250% of regular font size */
}

p {
  font-size: 20pt; /* size in points -- 20/72 of an inch */
}

.quote {
  font-size: smaller; /* smaller than normal size */
}

.bigger {
  font-size: 1.5em; /* 1.5 times larger than the 'M' in normal
font size */
}
```

Text Effects

There are numerous effects that can be added to text (or any element), many beyond the scope of this initial exploration of CSS. Here are a few simple examples to give you an idea

`text-shadow` allows a shadow to be added to text, giving it a 3-D style appearance. The value includes a colour, `x` and `y` offsets that determine the distance of the shadow from the text. Finally, we can also add a `blur-radius`, indicating how much to blur the shadow.

```
.shadow-text {
  text-shadow: 1px 1px 2px pink;
}
```

`text-overflow` can be used to determine what the browser should do when the amount of text exceeds the available space in a container (e.g. in a `<div>` or `<p>` that isn't wide enough). For example, we can specify that we want to `clip` the contents and not show any more, or we can automatically display `...`, the `ellipsis`.

```
<style>
  .movie-title {
    text-overflow: ellipsis;
  }
</style>
<span class="movie-title">Pirates of the Caribbean: The Curse of the Black Perl</span>
```

`background` Properties

Every element has a background that we can modify. We might, for example, want to specify that the background be a certain colour; or we might want to use an image, or even tile an image multiple times (like wallpaper to create a pattern); or we might want to create a gradient, from one colour to another. All of these options and more are possible using the `background` property.

```
div.error {
  background: red;
}
```

Styling Links

We can control the way that links (i.e., `<a>`) appear in our document. By default they will have a solid blue underline, and when visited, a purple solid underline. If you want to remove the underline, or change it's colour to match the theme of a page, we can do that using CSS `pseudo-classes`.

With *pseudo-classes* we can specify certain states for the elements in our selector, for example:

- `a:link` - a normal, unvisited link (normally blue underline)
- `a:visited` - a link the user has visited previously (normally purple underline)
- `a:hover` - a link when hovered with the mouse
- `a:active` - a link when it is clicked (i.e., while the mouse button is pressed)

> NOTE: pseudo-classes can be used with any element, but we mention them here in relation to styling links, since we often need them to deal with different states for a link.

Let's alter our links so that they use blue text, with no underline. However, when hovered, add back the underline:

```
a:link,
a:visited {
  text-decoration: none;
}

a:hover,
```

CSS and the DOM via JavaScript

We've been discussing CSS in the context of HTML, but we also need to explore how to work with it via JavaScript. The DOM provides us a number of methods for examining and changing the CSS styles associated with elements.

First, we can use a DOM element's `style` property. Doing so gives us access to the inline `style` attribute of the element. We can get or set particular CSS property values via the `style` element using *camelCase* versions of the CSS property names. For example, `background-color` becomes `backgroundColor`, while `width` remains `width`.

```
// Change the background colour of all paragraphs to red
var elems = document.querySelectorAll('p');
for (var i = 0, len = elems.length; i < len; i++) {
  elems[i].style.backgroundColor = 'red';
}
```

Usually we don't need (or want) to alter properties one by one via the DOM. Instead, it's more common to add or remove classes to elements, which pre-define a set of properties.

Similar to an element's `style` property, we can also use its `classList` property. It has a number of useful methods:

- `add()` - adds one (or more) class names to the element. If any of them are already present, they are ignored.
- `remove()` - removes one (or more) class names from the element.
- `toggle()` - toggles a class name on (adds it) or off (removes it), depending on the current state.

- `contains()` - checks if the specified class name is already defined for this element.
- `replace()` - replaces an old class name with the new one.

Using classes and `classList`, it's possible for us to define various states for our UI by creating multiple classes, and then add/remove them at runtime as the program runs and data changes.

Imagine you were creating a media player, and needed to show lists of songs and videos. Some of the media has been played by the user, and some is new. We can define classes for both, and then use JavaScript to apply the correct class to each:

```
<style>
  .media-played {
    background: gray;
  }

  .media-new {
    border: dashed red;
  }
</style>
...
<script>
  // Loop through an array of media objects, and set the class for
each one
  mediaItems.forEach(function (media) {
    var mediaElem = document.getElementById(media.id);
    if (media.played) {
      mediaElem.add('media-played');
    } else {
      mediaElem.add('media-new');
    }
  });
```

Exercise: Using Third-Party CSS Libraries

We've been focused on the mechanics of writing CSS ourselves, and this is an important skill. In addition, it's a good idea to know how to use third-party CSS libraries created by other developers. There are many pre-existing CSS libraries and frameworks we can use to help us create the web pages and apps we desire.

How to use Third-Party CSS

There is a general pattern to using any CSS library in your web page.

1. Find a library you want to use. We've listed a number of interesting ones below.
2. Read the documentation. Every library is different, and the "installation" and "usage" instructions will usually guide you on next steps. Get used to reading technical documentation, so that you can learn to solve your own problems.
3. Figure out which file or files you need to include in your HTML. This will typically include one or more `.css` files, and maybe `.js` ,fonts, etc. You will likely need to use `<link>` and `<script>` elements
4. See if the CSS library you want to use is available via a Content Delivery Network (CDN). Try searching for your chosen library on cdnjs or another CDN.
5. Read the docs for your library to see if you need to include any special markup, classes, or other info in your HTML file in order for things to work. CSS libraries operate on HTML, and sometimes they will expect it to be in a particular format.

Popular CSS Libraries

Here's a list of some popular CSS libraries and frameworks to get you started.

First, a few examples of simple "drop in" style libraries, where you simply include the CSS file, and everything "Just Works":

- Normalize.css - normalizes CSS so it is the same in all browsers (CDN link)
- Milligram - tiny set of default styles to make your site look great (CDN link)
- Tacit - CSS framework with no classes.

Next, there are lots of stylesheets you can use to improve the readability of your text:

- TufteCSS - a stylesheet based on the ideas of Edward Tufte about typography and text (CDN link)
- Gutenberg - a drop-in stylesheet for Printing to a printer
- Font Awesome - beautiful fonts and icons

In addition to changing how our text looks, a lot of CSS libraries add interesting and playful animations and effects to spice up our HTML:

- Hover - hover effects for links, buttons, and logs
- Balloon.css - tooltips and popups
- Animate.css - animations for HTML elements
- CSShake - more animations for HTML elements (CDN link)
- CSSgram - Instagram style filters for HTML images

Another common problem CSS can solve is what to do while we wait for things to finish loading:

- SpinKit - loading animations (CDN link)
- CSS Loader - more loading animations (CDN link)

Many CSS libraries have grown into more complex suites of layout, component, typography, navigation, and other solutions. We often refer to these as "frameworks" to indicate the expanded scope. There are many to choose from, including:

- Pure.css - tiny CSS framework for responsive layouts, buttons, forms, menus, etc.
- PaperCSS - playful, hand-drawn style UI kit
- Bootstrap - one of the most popular UI grid and component system for mobile and desktop web. Lots of themed versions of this too, for example Material UI
- UIKit - lightweight toolkit for building web app front-ends
- Semantic UI - UI framework, lots of responsive components (CDN link)
- Tailwind CSS - is a utility-first CSS framework for rapidly building modern websites without ever leaving your HTML.

TODO

TODO

TODO

TODO

Learning resources

Here you will find information about and links to learning resources that you will use in this course.

TODO: Finish this section