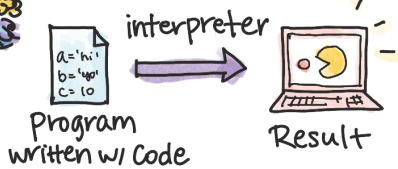


Introduction to Programming Languages and Tools of the Trade

This lesson covers the basics of programming languages. The topics covered here apply to most modern programming languages today. In the 'Tools of the Trade' section, you'll learn about useful software that helps you as a developer.

Introduction to Programming & Tools

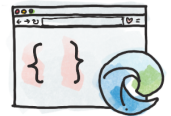


Tools

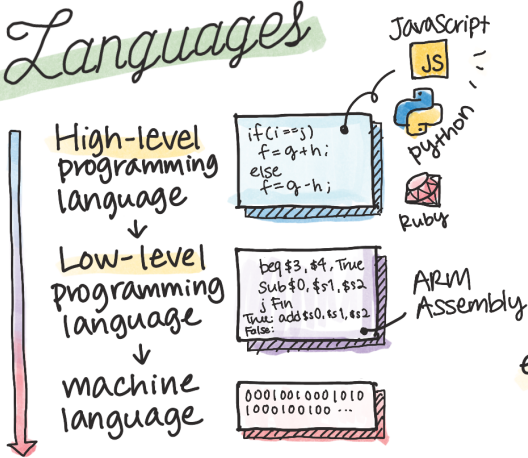
- ♥ Editors...
 - where you write code + debug. you may run the code too!
 - e.g. VSCode, Atom



- ♥ Browsers...
 - run your code on web
 - view visual elements
 - use DevTools to inspect + debug
 - e.g. Edge, Chrome, Firefox



Languages



♥ Command Line Tools...

- Send commands (lines of text) to execute tasks
- Less graphical option
- e.g. PowerShell, Terminal, Bash



♥ Docs...

- where you learn
- e.g. Mozilla Developer Network
- Frontend Masters

@azure-advocates
@girlie-mac

Sketchnote by [Tomomi Imura](#)

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

In this lesson, we'll cover:

- What is programming?
- Types of programming languages
- Basic elements of a program
- Useful software and tooling for the professional developer

You can take this lesson on [Microsoft Learn!](#)

What is Programming?

Programming (also known as coding) is the process of writing instructions to a device, such as a computer or mobile device. We write these instructions with a programming language, which is then interpreted by the device. These sets of instructions may be referred to by various names, but *program*, *computer program*, *application (app)*, and *executable* are a few popular names.

A *program* can be anything that is written with code; websites, games, and phone apps are programs. While it's possible to create a program without writing code, the underlying logic is interpreted to the device and that logic was most likely written with code. A program that is *running* or *executing code* is carrying out instructions. The device that you're currently reading this lesson with is running a program to print it to your screen.

✅ Do a little research: who is considered to have been the world's first computer programmer?

Programming Languages

Programming languages serve a main purpose: for developers to build instructions to send to a device. Devices only can understand binary (1s and 0s), and for *most* developers that's not a very efficient way to communicate. Programming languages are a vehicle for communication between humans and computers.

Programming languages come in different formats and may serve different purposes. For example, JavaScript is primarily used for web applications, while Bash is primarily used for operating systems.

Low level languages typically require fewer steps than *high level languages* for a device to interpret instructions. However, what makes high level languages popular is its readability and support.

JavaScript is considered a high level language.

The following code illustrates the difference between a high level language with JavaScript and low level language with ARM assembly code.

javascript

```
let number = 10
let n1 = 0, n2 = 1, nextTerm;

for (let i = 1; i <= number; i++) {
  console.log(n1);
  nextTerm = n1 + n2;
  n1 = n2;
  n2 = nextTerm;
}
```

c

```
area ascen,code,readonly
entry
code32
adr r0,thumb+1
bx r0
code16
thumb
mov r0,#00
sub r0,r0,#01
mov r1,#01
mov r4,#10
ldr r2,=0x40000000
back add r0,r1
str r0,[r2]
add r2,#04
mov r3,r0
mov r0,r1
mov r1,r3
sub r4,#01
cmp r4,#00
bne back
end
```

Believe it or not, *they're both doing the same thing*: printing a Fibonacci sequence up to 10.

✔ A Fibonacci sequence is defined as a set of numbers such that each number is the sum of the two preceding ones, starting from 0 and 1.

Elements of a program

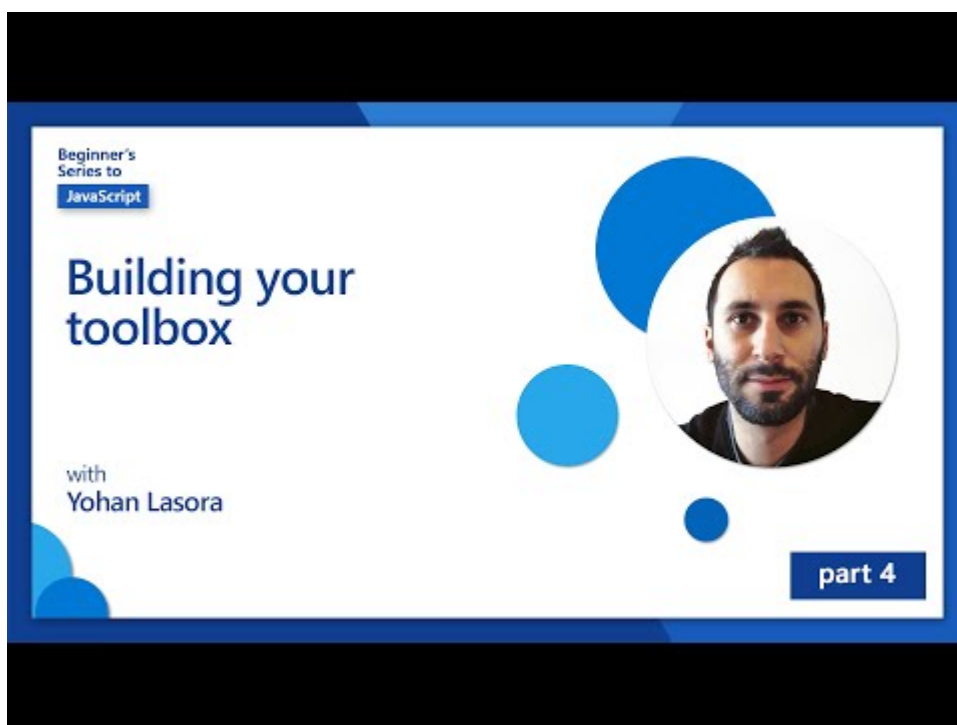
A single instruction in a program is called a *statement* and will usually have a character or line spacing that marks where the instruction ends, or *terminates*. How a program terminates varies with each language.

Most programs rely on using data from a user or elsewhere, where statements may rely on data to carry out instructions. Data can change how a program behaves, so programming languages come with a way to temporarily store data that can be used later. This data is called *variables*. Variables are statements that instruct a device to save data in its memory. Variables in programs are similar to ones in algebra, where they have a unique name and their value may change over time.

There's a chance that some statements will not be executed by a device. This is usually by design when written by the developer or by accident when an unexpected error occurs. This type of control of an application makes it more robust and maintainable. Typically these changes in control happen when certain decisions are met. A common statement in modern programming languages to control how a program is run is the `if..else` statement.

✔ You'll learn more about this type of statement in subsequent lessons

Tools of the Trade



 Click the image above for a video about tooling

In this section, you'll learn about some software that you might find very useful as you start your professional development journey.

A **development environment** is a unique set of tools and features that a developer will use often when writing software. Some of these tools have been customized for a developer specific needs, and may change over time if a developer changes priorities in work or personal projects, or when they use a different programming language. Development environments are as unique as the developers who use them.

Editors

One of the most crucial tools for software development is the editor. Editors are where you write your code and sometimes where you will run your code.

Developers rely on editors for a few additional reasons:

- *Debugging* Discovering bugs and errors by stepping through code, line by line. Some editors have debugging capabilities, or can be customized and added for specific programming languages.
- *Syntax highlighting* Adds colors and text formatting to code, makes it easier to read. Most editors allow customized syntax highlighting.
- *Extensions and Integrations* Additions that are specialized for developers, by developers, for access to additional tools that aren't built into the base editor. For example, many developers also need a way to document their code and explain how it works and will install a spell check extension to check for typos. Most of these additions are intended for use within a specific editor, and most editors come with a way to search for available extensions.
- *Customization* Most editors are extremely customizable, and each developer will have their own unique development environment that suits their needs. Many also allow developers to create their own extension.

Popular Editors and Web Development Extensions

- [Visual Studio Code](#)
 - [Code Spell Checker](#)
 - [Live Share](#)
 - [Prettier - Code formatter](#)
- [Atom](#)
 - [spell-check](#)
 - [teletype](#)

- [atom-beautify](#).

Browsers

Another crucial tool is the browser. Web developers rely on the browser to observe how their code runs on the web, it's also used to view visual elements of a web page that are written in the editor, like HTML.

Many browsers come with *developer tools* (DevTools) that contain a set of helpful features and information to assist developers to collect and capture important insights about their application. For example: If a web page has errors, it's sometimes helpful to know when they occurred. DevTools in a browser can be configured to capture this information.

Popular Browsers and DevTools

- [Edge](#)
- [Chrome](#)
- [Firefox](#)

Command Line Tools

Some developers prefer a less graphical view for their daily tasks and rely on the command line to achieve this. Developing code requires a significant amount of typing, and some developers prefer to not disrupt their flow on the keyboard and will use keyboard shortcuts to swap between desktop windows, work on different files, and use tools. Most tasks can be completed with a mouse, but one benefit of using the command line is that a lot can be done with command line tools without the need of swapping between the mouse and keyboard. Another benefit of the command line is that they're configurable and you can save your custom configuration, change it later, and also import it to new development machines. Because development environments are so unique to each developer, some will avoid using the command line, some will rely on it entirely, and some prefer a mix of the two.


Popular Command Line Options

Options for the command line will differ based on the operating system you use.


 = comes preinstalled on the operating system.

Windows


- [Powershell](#) 

- [Command Line](#) (also known as CMD) 
- [Windows Terminal](#)
- [mintty](#)


MacOS

- [Terminal](#) 
- [iTerm](#)
- [Powershell](#)

Linux

- [Bash](#) 
- [KDE Konsole](#)
- [Powershell](#)

Popular Command Line Tools

- [Git](#)  (on most operating systems)
- [NPM](#)
- [Yarn](#)

Documentation

When a developer wants to learn something new, they'll most likely turn to documentation to learn how to use it. Developers rely on documentation often to guide them through how to use tools and languages properly, and also to gain deeper knowledge of how it works.

Popular Documentation on Web Development

- [Mozilla Developer Network](#)
- [Frontend Masters](#)

✅ Do some research: Now that you know the basics of a web developer's environment, compare and contrast it with a web designer's environment.

Challenge

Compare some programming languages. What are some of the unique traits of JavaScript vs. Java?
How about COBOL vs. Go?

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study


Study a bit on the different languages available to the programmer. Try to write a line in one language, and then redo it in two others. What do you learn?

Assignment


[Reading the Docs](#)

Introduction to GitHub

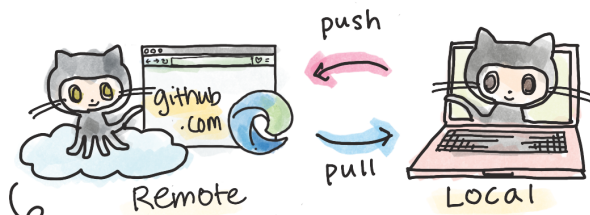
This lesson covers the basics of GitHub, a platform to host and manage changes to your code.

 Git = a distributed version control system for tracking changes in your code history

Introduction to GitHub

GitHub = a cloud-based hosting where you can manage your Git repositories 

★ Your first project with Git + GitHub ★



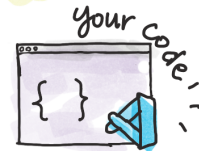
① Download + install Git
Then, set up your local Git profile

`git config --global ...`

① On github.com, Sign up / Sign in + Create a repository.



- ② Cd my project
- ③ git init
- ④ git status
- ⑤ git add.



- ⑨ To add more changes:
- ▶ git add.
 - ▶ git commit -m "typo fix"
 - ▶ git push

⑥ git commit -m "first commit"

⑦ git remote add origin http://github.com/-.git

⑧ git push -u origin main

@AzureAdvocates / @gitie_mac

Sketchnote by Tomomi Imura

Pre-Lecture Quiz

Pre-lecture quiz

Introduction

In this lesson, we'll cover:

- tracking the work you do on your machine
- working on projects with others
- how to contribute to open source software

Prerequisites

Before you begin, you'll need to check if Git is installed. In the terminal type: `git --version`

If Git is not installed, [download Git](#). Then, setup your local Git profile in the terminal:

- `git config --global user.name "your-name"`
- `git config --global user.email "your-email"`

To check if Git is already configured you can type: `git config --list`

You'll also need a GitHub account, a code editor (like Visual Studio Code), and you'll need to open your terminal (or: command prompt).

Navigate to github.com and create an account if you haven't already, or log in and fill out your profile.

✔️ GitHub isn't the only code repository in the world; there are others, but GitHub is the best known

Preparation

You'll need both a folder with a code project on your local machine (laptop or PC), and a public repository on GitHub, which will serve as an example for how to contribute to the projects of others.

Code management

Let's say you have a folder locally with some code project and you want to start tracking your progress using git - the version control system. Some people compare using git to writing a love letter to your future self. Reading your commit messages days or weeks or months later you'll be able to recall why you made a decision, or "rollback" a change - that is, when you write good "commit messages".

Task: Make a repository and commit code

1. **Create repository on GitHub.** On GitHub.com, in the repositories tab, or from the navigation bar top-right, find the **new repo** button.
 1. Give your repository (folder) a name
 2. Select **create repository**.
2. **Navigate to your working folder.** In your terminal, switch to the folder (also known as the directory) you want to start tracking. Type:

```
cd [name of your folder]
```

3. **Initialize a git repository.** In your project type:

bash

```
git init
```

4. **Check status.** To check the status of your repository type:

bash

```
git status
```

the output can look something like this:

output

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory
```

```
    modified:   file.txt
```

```
    modified:   file2.txt
```

Typically a `git status` command tells you things like what files are ready to be *saved* to the repo or has changes on it that you might want to persist.

5. **Add all files for tracking** This also called as staging files/ adding files to the staging area.

bash

```
git add .
```

The `git add` plus `.` argument indicates that all your files & changes for tracking.

6. **Add selected files for tracking**

bash

```
git add [file or folder name]
```

This helps us to add only selected files to the staging area when we don't want to commit all files at once.

7. **Unstage all files**

```
git reset
```

This command helps us to unstage all files at once.

8. Unstage a particular file

bash

```
git reset [file or folder name]
```

This command helps us to unstage only a particular file at once that we don't want to include for the next commit.

9. **Persisting your work.** At this point you've added the files to a so called *staging area*. A place where Git is tracking your files. To make the change permanent you need to *commit* the files. To do so you create a *commit* with the `git commit` command. A *commit* represents a saving point in the history of your repo. Type the following to create a *commit*:

bash

```
git commit -m "first commit"
```

This commits all of your files, adding the message "first commit". For future commit messages you will want to be more descriptive in your description to convey what type of change you've made.

10. **Connect your local Git repo with GitHub.** A Git repo is good on your machine but at some point you want to have backup of your files somewhere and also invite other people to work with you on your repo. One such great place to do so is GitHub. Remember we've already created a repo on GitHub so the only thing we need to do is to connect our local Git repo with GitHub. The command `git remote add` will do just that. Type the following command:

Note, before you type the command go to your GitHub repo page to find the repository URL. You will use it in the below command. Replace `repository_name` with your GitHub URL.

bash

```
git remote add origin https://github.com/username/repository_name.git
```

This creates a *remote*, or connection, named "origin" pointing at the GitHub repository you created earlier.

11. **Send local files to GitHub.** So far you've created a *connection* between the local repo and the GitHub repo. Let's send these files to GitHub with the following command `git push`, like so:

bash

```
git push -u origin main
```

This sends your commits in your "main" branch to GitHub.

12. **To add more changes.** If you want to continue making changes and pushing them to GitHub you'll just need to use the following three commands:

bash

```
git add .  
git commit -m "type your commit message here"  
git push
```

Tip, You might also want to adopt a `.gitignore` file to prevent files you don't want to track from showing up on GitHub - like that notes file you store in the same folder but has no place on a public repository. You can find templates for `.gitignore` files at [.gitignore templates](#).

Commit messages

A great Git commit subject line completes the following sentence: If applied, this commit will

For the subject use the imperative, present tense: "change" not "changed" nor "changes". As in the subject, in the body (optional) also use the imperative, present tense. The body should include the motivation for the change and contrast this with previous behavior. You're explaining the *why*, not the *how*.

✅ Take a few minutes to surf around GitHub. Can you find a really great commit message? Can you find a really minimal one? What information do you think is the most important and useful to convey in a commit message?

Task: Collaborate

The main reason for putting things on GitHub was to make it possible to collaborate with other developers.

Working on projects with others

In your repository, navigate to `Insights > Community` to see how your project compares to recommended community standards.

Here are some things that can improve your GitHub repo:

- **Description.** Did you add a description for your project?
- **README.** Did you add a README? GitHub provides guidance for writing a [README](#).
- **Contributing guideline.** Does your project have [contributing guidelines](#),
- **Code of Conduct.** a [Code of Conduct](#),
- **License.** Perhaps most importantly, a [license](#)?

All these resources will benefit onboarding new team members. And those are typically the kind of things new contributors look at before even looking at your code, to find out if your project is the right place for them to be spending their time.

✅ README files, although they take time to prepare, are often neglected by busy maintainers. Can you find an example of a particularly descriptive one? Note: there are some [tools to help create good READMEs](#) that you might like to try.

Task: Merge some code

Contributing docs help people contribute to the project. It explains what types of contributions you're looking for and how the process works. Contributors will need to go through a series of steps to be able to contribute to your repo on GitHub:

1. **Forking your repo** You will probably want people to *fork* your project. Forking means creating a replica of your repository on their GitHub profile.
2. **Clone.** From there they will clone the project to their local machine.
3. **Create a branch.** You will want to ask them to create a *branch* for their work.
4. **Focus their change on one area.** Ask contributors to concentrate their contributions on one thing at a time - that way the chances that you can *merge* in their work is higher. Imagine they write a bug fix, add a new feature, and update several tests - what if you want to, or can only implement 2 out of 3, or 1 out of 3 changes?

✅ Imagine a situation where branches are particularly critical to writing and shipping good code. What use cases can you think of?

Note, be the change you want to see in the world, and create branches for your own work as well. Any commits you make will be made on the branch you're currently "checked out" to.

Use `git status` to see which branch that is.

Let's go through a contributor workflow. Assume the contributor has already *forked* and *cloned* the repo so they have a Git repo ready to be worked on, on their local machine:

1. **Create a branch.** Use the command `git branch` to create a branch that will contain the changes they mean to contribute:

```
git branch [branch-name]
```

bash

2. **Switch to working branch.** Switch to the specified branch and update the working directory with `git checkout` :

```
git checkout [branch-name]
```

bash

3. **Do work.** At this point you want to add your changes. Don't forget to tell Git about it with the following commands:

```
git add .  
git commit -m "my changes"
```

bash

Ensure you give your commit a good name, for your sake as well as the maintainer of the repo you are helping on.

4. **Combine your work with the `main` branch.** At some point you are done working and you want to combine your work with that of the `main` branch. The `main` branch might have changed meanwhile so make sure you first update it to the latest with the following commands:

```
git checkout main  
git pull
```

bash

At this point you want to make sure that any *conflicts*, situations where Git can't easily *combine* the changes happens in your working branch. Therefore run the following commands:

```
git checkout [branch_name]  
git merge main
```

bash

This will bring in all changes from `main` into your branch and hopefully you can just continue. If not, VS Code will tell you where Git is *confused* and you just alter the affected files to say which content is the most accurate.

5. **Send your work to GitHub.** Sending your work to GitHub means two things. Pushing your branch to your repo and then open up a PR, Pull Request.

bash

```
git push --set-upstream origin [branch-name]
```

The above command creates the branch on your forked repo.

6. **Open a PR.** Next, you want to open up a PR. You do that by navigating to the forked repo on GitHub. You will see an indication on GitHub where it asks whether you want to create a new PR, you click that and you are taken to an interface where you can change commit message title, give it a more suitable description. Now the maintainer of the repo you forked will see this PR and *fingers crossed* they will appreciate and *merge* your PR. You are now a contributor, yay :)
7. **Clean up.** It's considered good practice to *clean up* after you successfully merge a PR. You want to clean up both your local branch and the branch you pushed to GitHub. First let's delete it locally with the following command:

bash

```
git branch -d [branch-name]
```

Ensure you go the GitHub page for the forked repo next and remove the remote branch you just pushed to it.

`Pull request` seems like a silly term because really you want to push your changes to the project. But the maintainer (project owner) or core team needs to consider your changes before merging it with the project's "main" branch, so you're really requesting a change decision from a maintainer.

A pull request is the place to compare and discuss the differences introduced on a branch with reviews, comments, integrated tests, and more. A good pull request follows roughly the same rules as a commit message. You can add a reference to an issue in the issue tracker, when your work for instance fixes an issue. This is done using a `#` followed by the number of your issue. For example `#97` .

👉 Fingers crossed that all checks pass and the project owner(s) merge your changes into the project
👉

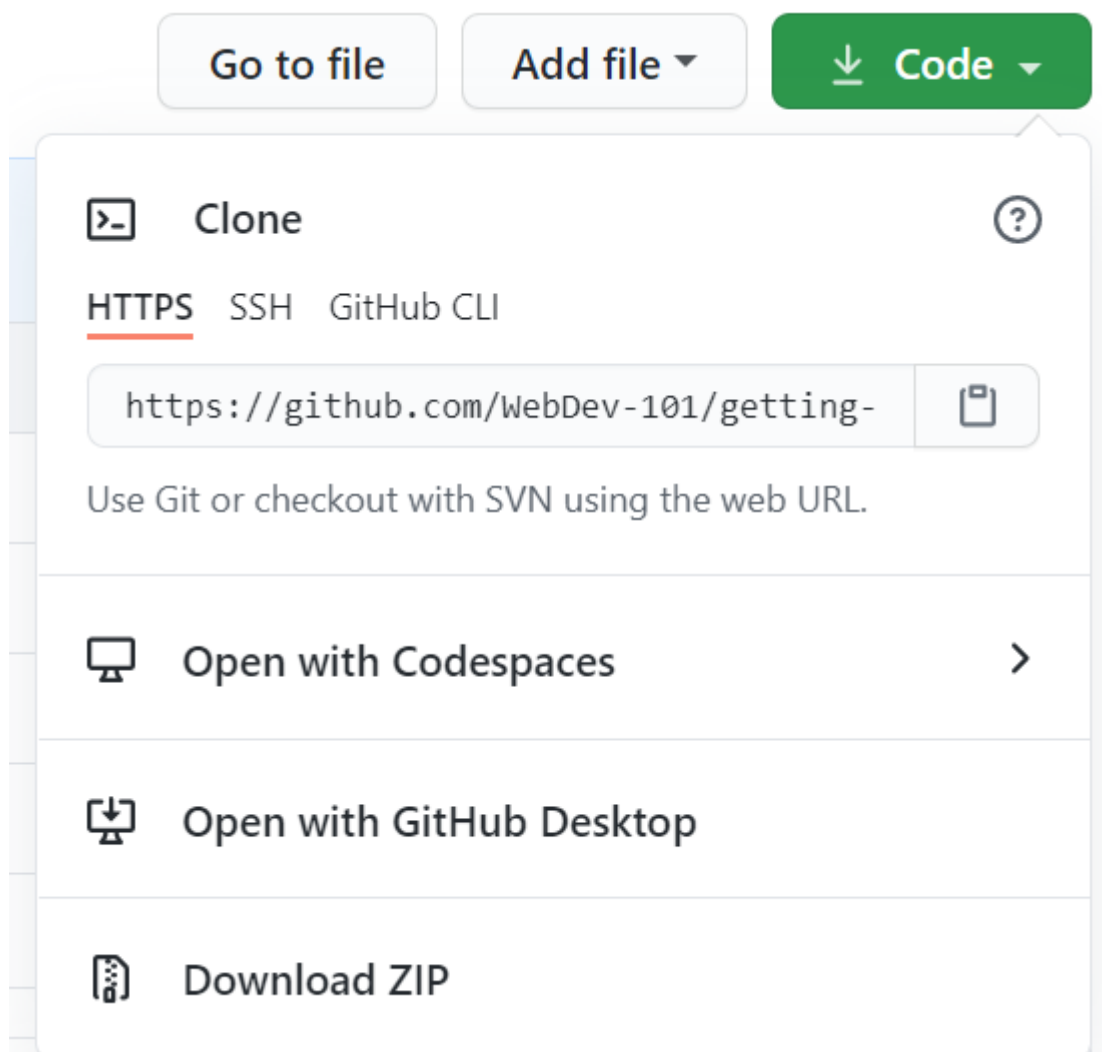
Update your current local working branch with all new commits from the corresponding remote branch on GitHub:

git pull

How to contribute to open source

First, let's find a repository (or **repo**) on GitHub of interest to you and to which you'd like to contribute a change. You will want to copy its contents to your machine.

✓ A good way to find 'beginner-friendly' repos is to [search by the tag 'good-first-issue'](#).



There are several ways of copying code. One way is to "clone" the contents of the repository, using HTTPS, SSH, or using the GitHub CLI (Command Line Interface).

Open your terminal and clone the repository like so:

```
git clone https://github.com/ProjectURL
```

To work on the project, switch to the right folder: `cd ProjectURL`

You can also open the entire project using [Codespaces](#), GitHub's embedded code editor / cloud development environment, or [GitHub Desktop](#).

Lastly, you can download the code in a zipped folder.

A few more interesting things about GitHub

You can star, watch and/or "fork" any public repository on GitHub. You can find your starred repositories in the top-right drop-down menu. It's like bookmarking, but for code.

Projects have an issue tracker, mostly on GitHub in the "Issues" tab unless indicated otherwise, where people discuss issues related to the project. And the Pull Requests tab is where people discuss and review changes that are in progress.

Projects might also have discussion in forums, mailing lists, or chat channels like Slack, Discord or IRC.

✔ Take a look around your new GitHub repo and try a few things, like editing settings, adding information to your repo, and creating a project (like a Kanban board). There's a lot you can do!

Challenge

Pair with a friend to work on each other's code. Create a project collaboratively, fork code, create branches, and merge changes.

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Read more about [contributing to open source software](#).

[Git cheatsheet](#).

Practice, practice, practice. GitHub has great learning paths available via [lab.github.com](#):

- [First Week on GitHub](#)

You'll also find more advanced labs.

Assignment

Complete [the First Week on GitHub training lab](#)

Creating Accessible Webpages

@azureadvocates @girlie_mac

Creating Accessible Webpages

Tools

- ✓ Screen readers
- ✓ Contrast checkers
- ✓ Lighthouse in Devtools

JAWS

WCAG color checker

Designing

- ✓ Color safe palette
- ✓ Semantic HTML
- ✓ `...` to emphasize
- ✗ ``
- ✓ Visual clues

Univ. of Minnesota

Accessible U

Hyperlinks


- ✓ Let screen reader read out hyperlinks
- ✓ "Visit [Wikipedia.org](#) for more info."
- ✗ "Click here for more info."

Accessible links = Better SEO

ARIA


Accessible Rich Internet Applications
a set of HTML attributes

```
<div role="progressbar" aria-valuenow="75">
```



Images


- ✓ alt attribute



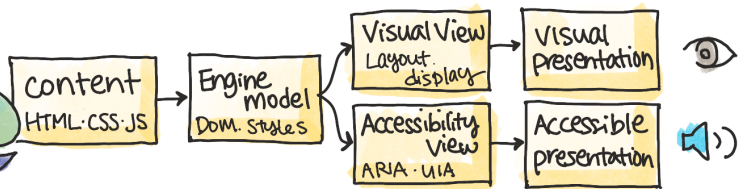
alt="Bit the Raccoon"

Keyboard

- ✓ Keyboard navigation



Diagram



```
graph LR; Content[content HTML, CSS, JS] --> Engine[Engine Model DOM, styles]; Engine --> VisualView[Visual View Layout, display]; Engine --> AccessibilityView[Accessibility View ARIA, UIA]; VisualView --> VisualPresentation[Visual presentation]; AccessibilityView --> AccessiblePresentation[Accessible presentation];
```

Sketchnote by [Tomomi Imura](#)

Pre-Lecture Quiz

[Pre-lecture quiz](#)

The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect.

This quote perfectly highlights the importance of creating accessible websites. An application that can't be accessed by all is by definition exclusionary. As web developers we should always have accessibility in mind. By having this focus from the beginning you will be well on your way to ensure everyone can access the pages you create. In this lesson, you'll learn about the tools that can help you ensure that your web assets are accessible and how to build with accessibility in mind.

You can take this lesson on [Microsoft Learn!](#)

Tools to use

Screen readers

One of the best-known accessibility tools are screen readers.

Screen readers are commonly used clients for those with vision impairments. As we spend time ensuring a browser properly conveys the information we wish to share, we must also ensure a screen reader does the same.

At its most basic, a screen reader will read a page from top to bottom audibly. If your page is all text, the reader will convey the information in a similar fashion to a browser. Of course, web pages are rarely purely text; they will contain links, graphics, color, and other visual components. Care must be taken to ensure that this information is read correctly by a screen reader.

Every web developer should familiarize themselves with a screen reader. As highlighted above, it's the client your users will utilize. Much in the same way you're familiar with how a browser operates, you should learn how a screen reader operates. Fortunately, screen readers are built into most operating systems.

Some browsers also have built-in tools and extensions that can read text aloud or even provide some basic navigational features, such as [these accessibility-focused Edge browser tools](#). These are also important accessibility tools, but function very differently from screen readers and they should not be mistaken for screen reader testing tools.

✔ Try a screen reader and browser text reader. On Windows [Narrator](#) is included by default, and [JAWS](#) and [NVDA](#) can also be installed. On macOS and iOS, [VoiceOver](#) is installed by default.

Zoom

Another tool commonly used by people with vision impairments is zooming. The most basic type of zooming is static zoom, controlled through `Control + plus sign (+)` or by decreasing screen resolution. This type of zoom causes the entire page to resize, so using [responsive design](#) is important to provide a good user experience at increased zoom levels.

Another type of zoom relies on specialized software to magnify one area of the screen and pan, much like using a real magnifying glass. On Windows, [Magnifier](#) is built in, and [ZoomText](#) is a third-party magnification software with more features and a larger user base. Both macOS and iOS have a built-in magnification software called [Zoom](#).

Contrast checkers

Colors on web sites need to be carefully chosen to answer the needs of color-blind users or people who have difficulty seeing low-contrast colors.

✅ Test a web site you enjoy using for color usage with a browser extension such as [WCAG's color checker](#). What do you learn?

Lighthouse

In the developer tool area of your browser, you'll find the Lighthouse tool. This tool is important to get a first view of the accessibility (as well as other analysis) of a web site. While it's important not to rely exclusively on Lighthouse, a 100% score is very helpful as a baseline.

✅ Find Lighthouse in your browser's developer tool panel and run an analysis on any site. what do you discover?

Designing for accessibility

Accessibility is a relatively large topic. To help you out, there are numerous resources available.

- [Accessible U - University of Minnesota](#)

While we won't be able to cover every aspect of creating accessible sites, below are some of the core tenets you will want to implement. Designing an accessible page from the start is **always** easier than going back to an existing page to make it accessible.

Good display principles

Color safe palettes

People see the world in different ways, and this includes colors. When selecting a color scheme for your site, you should ensure it's accessible to all. One great [tool for generating color palettes is Color Safe](#).

✅ Identify a web site that is very problematic in its use of color. Why?

Use the correct HTML

With CSS and JavaScript it's possible to make any element look like any type of control. `` could be used to create a `<button>`, and `` could become a hyperlink. While this might be considered easier to style, it conveys nothing to a screen reader. Use the appropriate HTML when creating controls on a page. If you want a hyperlink, use `<a>`. Using the right HTML for the right control is called making use of Semantic HTML.

✅ Go to any web site and see if the designers and developers are using HTML properly. Can you find a button that should be a link? Hint: right click and choose 'View Page Source' in your browser to look at underlying code.

Create a descriptive heading hierarchy

Screen reader users [rely heavily on headings](#) to find information and navigate through a page. Writing descriptive heading content and using semantic heading tags are important for creating an easily navigable site for screen reader users.

Use good visual clues

CSS offers complete control over the look of any element on a page. You can create text boxes without an outline or hyperlinks without an underline. Unfortunately removing those clues can make it more challenging for someone who depends on them to be able to recognize the type of control.

The importance of link text

Hyperlinks are core to navigating the web. As a result, ensuring a screen reader can properly read links allows all users to navigate your site.

Screen readers and links

As you would expect, screen readers read link text in the same way they'd read any other text on the page. With this in mind, the text demonstrated below might feel perfectly acceptable.

The little penguin, sometimes known as the fairy penguin, is the smallest penguin in the world.
[Click here](#) for more information.

The little penguin, sometimes known as the fairy penguin, is the smallest penguin in the world.
Visit https://en.wikipedia.org/wiki/Little_penguin for more information.

NOTE As you're about to read, you should **never** create links which look like the above.

Remember, screen readers are a different interface from browsers with a different set of features.

The problem with using the URL

Screen readers read the text. If a URL appears in the text, the screen reader will read the URL. Generally speaking, the URL does not convey meaningful information, and can sound annoying. You may have experienced this if your phone has ever audibly read a text message with a URL.

The problem with "click here"

Screen readers also have the ability to read only the hyperlinks on a page, much in the same way a sighted person would scan a page for links. If the link text is always "click here", all the user will hear is "click here, click here, click here, click here, click here, ..." All links are now indistinguishable from one another.

Good link text

Good link text briefly describes what's on the other side of the link. In the above example talking about little penguins, the link is to the Wikipedia page about the species. The phrase *little penguins* would make for perfect link text as it makes it clear what someone will learn about if they click the link - little penguins.

The little penguin, sometimes known as the fairy penguin, is the smallest penguin in the world.

- ✓ Surf the web for a few minutes to find pages that use obscure linking strategies. Compare them with other, better-linked sites. What do you learn?

Search engine notes

As an added bonus for ensuring your site is accessible to all, you'll help search engines navigate your site as well. Search engines use link text to learn the topics of pages. So using good link text helps everyone!

ARIA

Imagine the following page:

Product	Description	Order
Widget	<u>Description</u>	<u>Order</u>
Super widget	<u>Description</u>	<u>Order</u>

In this example, duplicating the text of description and order make sense for someone using a browser. However, someone using a screen reader would only hear the words *description* and *order* repeated without context.

To support these types of scenarios, HTML supports a set of attributes known as Accessible Rich Internet Applications (ARIA). These attributes allow you to provide additional information to screen readers.

NOTE: Like many aspects of HTML, browser and screen reader support may vary. However, most mainline clients support ARIA attributes.

You can use `aria-label` to describe the link when the format of the page doesn't allow you to. The description for widget could be set as

html

```
<a href="#" aria-label="Widget description">description</a>
```


✔ In general, using Semantic markup as described above supersedes the use of ARIA, but sometimes there is no semantic equivalent for various HTML widgets. A good example is a Tree. There's no HTML equivalent for a tree, so you identify the generic `<div>` for this element with a proper role and aria values. The [MDN documentation on ARIA](#) contains more useful information.

html

```
<h2 id="tree-label">File Viewer</h2>
<div role="tree" aria-labelledby="tree-label">
  <div role="treeitem" aria-expanded="false" tabindex="0">Uploads</div>
</div>
```

Images

It goes without saying screen readers are unable to automatically read what's in an image. Ensuring images are accessible doesn't take much work - it's what the `alt` attribute is all about. All meaningful images should have an `alt` to describe what they are. Images that are purely decorative should have their `alt` attribute set to an empty string: `alt=""`. This prevents screen readers from unnecessarily announcing the decorative image.

✔ As you might expect, search engines are also unable to understand what's in an image. They also use alt text. So once again, ensuring your page is accessible provides additional bonuses!

The keyboard

Some users are unable to use a mouse or trackpad, instead relying on keyboard interactions to tab from one element to the next. It's important for your web site to present your content in logical order so a keyboard user can access each interactive element as they move down a document. If you build your web pages with semantic markup and use CSS to style their visual layout, your site should be keyboard-navigable, but it's important to test this aspect manually. Learn more about [keyboard navigation strategies](#).

✔ Go to any web site and try to navigate through it using only your keyboard. What works, what doesn't work? Why?

Summary

A web accessible to some is not a truly 'world-wide web'. The best way to ensure the sites you create are accessible is to incorporate accessibility best practices from the start. While there are extra steps involved, incorporating these skills into your workflow now will mean all pages you create will be accessible.

Challenge

Take this HTML and rewrite it to be as accessible as possible, given the strategies you learned.

html

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      Example
    </title>
    <link href='../assets/style.css' rel='stylesheet' type='text/css'>
  </head>
  <body>
    <div class="site-header">
      <p class="site-title">Turtle Ipsum</p>
      <p class="site-subtitle">The World's Premier Turtle Fan Club</p>
    </div>
    <div class="main-nav">
      <p class="nav-header">Resources</p>
      <div class="nav-list">
        <p class="nav-item nav-item-bull"><a href="https://www.youtube.com,
        <p class="nav-item nav-item-bull"><a href="https://en.wikipedia.org
        <p class="nav-item nav-item-bull"><a href="https://en.wikipedia.org
      </div>
    </div>
    <div class="main-content">
      <div>
        <p class="page-title">Welcome to Turtle Ipsum.
          <a href="">Click here</a> to learn more.
        </p>
        <p class="article-text">
          Turtle ipsum dolor sit amet, consectetur adipiscing elit, sed do
        </p>
      </div>
    </div>
  </div>
</div>
```

```
<div class="footer">
  <div class="footer-section">
    <span class="button">Sign up for turtle news</span>
  </div><div class="footer-section">
    <p class="nav-header footer-title">
      Internal Pages
    </p>
    <div class="nav-list">
      <p class="nav-item nav-item-bull"><a href="..">Index</a></p>
      <p class="nav-item nav-item-bull"><a href="../semantic">Semantic
    </div>
  </div>
  <p class="footer-copyright">&copy; 2016 Instrument</span>
</div>
</body>
</html>
```

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Many governments have laws regarding accessibility requirements. Read up on your home country's accessibility laws. What is covered, and what isn't? An example is [this government web site](#).

Assignment

[Analyze a non-accessible web site](#)

Credits: [Turtle Ipsum](#) by Instrument

JavaScript Basics: Data Types

Variables

* 3 different keywords

var

- Function scoped
- can be changed in scope
- Avail. before declared!

let

- Block scoped
- can be changed in scope
- Only avail after declaration

const

- Block scoped
- cannot be changed
- only avail after declaration

@AzureAdvocates / @girlie_mac

keyword → `const` variable name → `greeting = "Hello";`
 Declaring a variable → assigned value

Data types

String
 a set of characters that reside between single or double quotes. ☺☺

Number
`let donut = 32;`

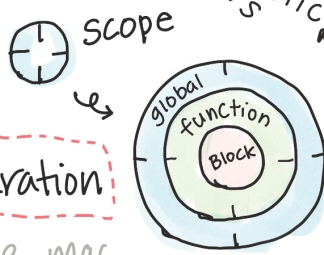
can be: integer, negative decimals, etc.
 also: Infinity, BigInt

JavaScript Basics Data Types



- + Addition
- Subtraction
- * Multiplication
- / division
- % Remainder

Boolean



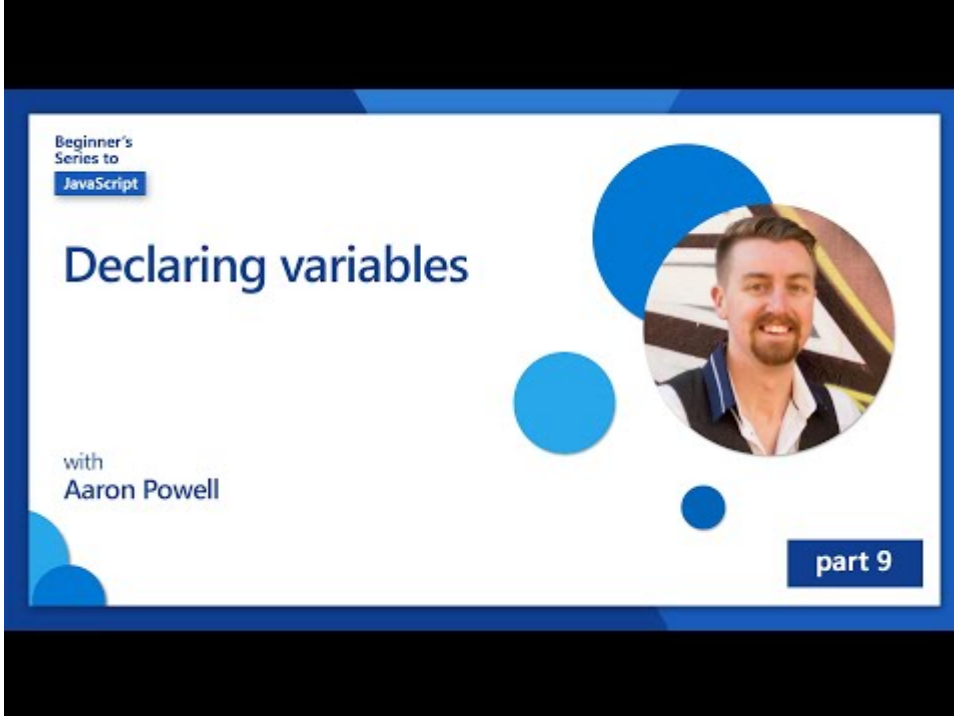
Sketchnote by [Tomomi Imura](#)

Pre-Lecture Quiz

Pre-lecture quiz

This lesson covers the basics of JavaScript, the language that provides interactivity on the web.

You can take this lesson on [Microsoft Learn!](#)



 Click the image above for a video about data types

Let's start with variables and the data types that populate them!

Variables

Variables store values that can be used and changed throughout your code.

Creating and **declaring** a variable has the following syntax **[keyword] [name]**. It's made up of the two parts:

- **Keyword.** Keywords can be `let` or `var`.

Note, The keyword `let` was introduced in ES6 and gives your variable a so called block scope. It's recommended that you use `let` over `var`. We will cover block scopes more in depth in future parts.

- **The variable name,** this is a name you choose yourself.

Task - working with variables

1. **Declare a variable.** Let's declare a variable using the `let` keyword:

javascript

```
let myVariable;
```

`myVariable` has now been declared using the `let` keyword. It currently doesn't have a value.

2. **Assign a value.** Store a value in a variable with the `=` operator, followed by the expected value.

javascript

```
myVariable = 123;
```

Note: the use of `=` in this lesson means we make use of an "assignment operator", used to set a value to a variable. It doesn't denote equality.

`myVariable` has now been *initialized* with the value 123.

3. **Refactor.** Replace your code with the following statement.

javascript

```
let myVariable = 123;
```

The above is called an *explicit initialization* when a variable is declared and is assigned a value at the same time.

4. **Change the variable value.** Change the variable value in the following way:

javascript

```
myVariable = 321;
```

Once a variable is declared, you can change its value at any point in your code with the `=` operator and the new value.

✔ Try it! You can write JavaScript right in your browser. Open a browser window and navigate to Developer Tools. In the console, you will find a prompt; type `let myVariable = 123`, press return, then type `myVariable`. What happens? Note, you'll learn more about these concepts in subsequent lessons.

Constants

Declaration and initialization of a constant follows the same concepts as a variable, with the exception of the `const` keyword. Constants are typically declared with all uppercase letters.

javascript

```
const MY_VARIABLE = 123;
```

Constants are similar to variables, with two exceptions:

- **Must have a value.** Constants must be initialized, or an error will occur when running code.
- **Reference cannot be changed.** The reference of a constant cannot be changed once initialized, or an error will occur when running code. Let's look at two examples:
 - **Simple value.** The following is NOT allowed:

javascript

```
const PI = 3;  
PI = 4; // not allowed
```

- **Object reference is protected.** The following is NOT allowed.

javascript

```
const obj = { a: 3 };  
obj = { b: 5 } // not allowed
```

- **Object value is not protected.** The following IS allowed:

javascript

```
const obj = { a: 3 };  
obj.a = 5; // allowed
```

Above you are changing the value of the object but not the reference itself, which makes it allowed.

Note, a `const` means the reference is protected from reassignment. The value is not immutable though and can change, especially if it's a complex construct like an object.

Data Types

Variables can store many different types of values, like numbers and text. These various types of values are known as the **data type**. Data types are an important part of software development because it helps developers make decisions on how the code should be written and how the software should run. Furthermore, some data types have unique features that help transform or extract additional information in a value.

✅ Data Types are also referred to as JavaScript data primitives, as they are the lowest-level data types that are provided by the language. There are 6 primitive data types: string, number, bigint, boolean, undefined, and symbol. Take a minute to visualize what each of these primitives might represent. What is a zebra ? How about 0 ? true ?

Numbers

In the previous section, the value of `myVariable` was a number data type.

```
let myVariable = 123;
```

Variables can store all types of numbers, including decimals or negative numbers. Numbers also can be used with arithmetic operators, covered in the [next section](#).

Arithmetic Operators

There are several types of operators to use when performing arithmetic functions, and some are listed here:

Symbol	Description	Example
+	Addition: Calculates the sum of two numbers	<code>1 + 2 //expected answer is 3</code>
-	Subtraction: Calculates the difference of two numbers	<code>1 - 2 //expected answer is -1</code>
*	Multiplication: Calculates the product of two numbers	<code>1 * 2 //expected answer is 2</code>
/	Division: Calculates the quotient of two numbers	<code>1 / 2 //expected answer is 0.5</code>

Symbol	Description	Example
<code>%</code>	Remainder: Calculates the remainder from the division of two numbers	<code>1 % 2 //expected answer is 1</code>

✔ Try it! Try an arithmetic operation in your browser's console. Do the results surprise you?

Strings

Strings are sets of characters that reside between single or double quotes.

- `'This is a string'`
- `"This is also a string"`
- `let myString = 'This is a string value stored in a variable';`

Remember to use quotes when writing a string, or else JavaScript will assume it's a variable name.

Formatting Strings

Strings are textual, and will require formatting from time to time.

To **concatenate** two or more strings, or join them together, use the `+` operator.

javascript

```
let myString1 = "Hello";
let myString2 = "World";

myString1 + myString2 + "!"; //HelloWorld!
myString1 + " " + myString2 + "!"; //Hello World!
myString1 + ", " + myString2 + "!"; //Hello, World!
```

✔ Why does `1 + 1 = 2` in JavaScript, but `'1' + '1' = 11`? Think about it. What about `'1' + 1`?

Template literals are another way to format strings, except instead of quotes, the backtick is used. Anything that is not plain text must be placed inside placeholders `${ }`. This includes any variables that may be strings.

javascript

```
let myString1 = "Hello";
let myString2 = "World";
```

```
`${myString1} ${myString2}!` //Hello World!  
`${myString1}, ${myString2}!` //Hello, World!
```

You can achieve your formatting goals with either method, but template literals will respect any spaces and line breaks.

✅ When would you use a template literal vs. a plain string?

Booleans

Booleans can be only two values: `true` or `false`. Booleans can help make decisions on which lines of code should run when certain conditions are met. In many cases, operators assist with setting the value of a Boolean and you will often notice and write variables being initialized or their values being updated with an operator.

- `let myTrueBool = true`
- `let myFalseBool = false`

✅ A variable can be considered 'truthy' if it evaluates to a boolean `true`. Interestingly, in JavaScript, all values are truthy unless defined as falsy.

Challenge

JavaScript is notorious for its surprising ways of handling datatypes on occasion. Do a bit of research on these 'gotchas'. For example: case sensitivity can bite! Try this in your console:

```
let age = 1; let Age = 2; age == Age (resolves false -- why?). What other gotchas can you find?
```

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Take a look at [this list of JavaScript exercises](#) and try one. What did you learn?

Assignment

Data Types Practice

JavaScript Basics: Methods and Functions

Function: a building block of code that we can execute on demand.

- ♥ Take an input
- ♥ Return an output

Declaration *function name* *parameter(s)*

```
function square(n) {  
  return n * n;  
}
```

Return an output

☆ Passing info

```
function juice(  
  (apple, orange)  
) {  
}
```

☆ Default values

```
function displayGreet(name, sal = 'Hello') {  
  console.log(`${sal}, ${name}`);  
}
```

☆ Function as parameter

```
function displayDone() {  
  console.log('3 sec. elapsed.');
```

☆ Anonymous Function

```
setTimeout(3000, function() {  
  console.log(-----);  
});
```

☆ Fat Arrow Function

```
setTimeout(3000, () => {  
  console.log(-----);  
});
```

Const myNum = square(25);

JavaScript Basics
Functions

rewrite

ES6

@AzureAdvocates / @girlie_mac

Sketchnote by [Tomomi Imura](#)

Pre-Lecture Quiz

Pre-lecture quiz

When we think about writing code, we always want to ensure our code is readable. While this sounds counterintuitive, code is read many more times than it's written. One core tool in a developer's toolbox to ensure maintainable code is the **function**.



 Click the image above for a video about methods and functions.

You can take this lesson on [Microsoft Learn!](#)

Functions

At its core, a function is a block of code we can execute on demand. This is perfect for scenarios where we need to perform the same task multiple times; rather than duplicating the logic in multiple locations (which would make it hard to update when the time comes), we can centralize it in one location, and call it whenever we need the operation performed - you can even call functions from other functions!.

Just as important is the ability to name a function. While this might seem trivial, the name provides a quick way of documenting a section of code. You could think of this as a label on a button. If I click on a button which reads "Cancel timer", I know it's going to stop running the clock.

Creating and calling a function

The syntax for a function looks like the following:

```
function nameOfFunction() { // function definition
  // function definition/body
}
```

If I wanted to create a function to display a greeting, it might look like this:

javascript

```
function displayGreeting() {
  console.log('Hello, world!');
}
```

Whenever we want to call (or invoke) our function, we use the name of the function followed by `()`. It's worth noting the fact our function can be defined before or after we decide to call it; the JavaScript compiler will find it for you.

javascript

```
// calling our function
displayGreeting();
```

NOTE: There is a special type of function known as a **method**, which you've already been using! In fact, we saw this in our demo above when we used `console.log`. What makes a method different from a function is a method is attached to an object (`console` in our example), while a function is free floating. You will hear many developers use these terms interchangeably.

Function best practices

There are a handful of best practices to keep in mind when creating functions

- As always, use descriptive names so you know what the function will do
- Use **camelCasing** to combine words
- Keep your functions focused on a specific task

Passing information to a function

To make a function more reusable you'll often want to pass information into it. If we consider our `displayGreeting` example above, it will only display **Hello, world!**. Not the most useful function one could create. If we want to make it a little more flexible, like allowing someone to specify the name of the person to greet, we can add a **parameter**. A parameter (also sometimes called an **argument**), is additional information sent to a function.

Parameters are listed in the definition part within parenthesis and are comma separated like so:

javascript

```
function name(param, param2, param3) {  
  
}
```

We can update our `displayGreeting` to accept a name and have that displayed.

javascript

```
function displayGreeting(name) {  
  const message = `Hello, ${name}!`;  
  console.log(message);  
}
```

When we want to call our function and pass in the parameter, we specify it in the parenthesis.

javascript

```
displayGreeting('Christopher');  
// displays "Hello, Christopher!" when run
```

Default values

We can make our function even more flexible by adding more parameters. But what if we don't want to require every value be specified? Keeping with our greeting example, we could leave name as required (we need to know who we're greeting), but we want to allow the greeting itself to be customized as desired. If someone doesn't want to customize it, we provide a default value instead. To provide a default value to a parameter, we set it much in the same way we set a value for a variable - `parameterName = 'defaultValue'` . To see a full example:

javascript

```
function displayGreeting(name, salutation='Hello') {  
  console.log(`${salutation}, ${name}`);  
}
```

When we call the function, we can then decide if we want to set a value for `salutation` .

javascript

```
displayGreeting('Christopher');  
// displays "Hello, Christopher"  
  
displayGreeting('Christopher', 'Hi');  
// displays "Hi, Christopher"
```

Return values

Up until now the function we built will always output to the console. Sometimes this can be exactly what we're looking for, especially when we create functions which will be calling other services. But what if I want to create a helper function to perform a calculation and provide the value back so I can use it elsewhere?

We can do this by using a **return value**. A return value is returned by the function, and can be stored in a variable just the same as we could store a literal value such as a string or number.

If a function does return something then the keyword `return` is used. The `return` keyword expects a value or reference of what's being returned like so:

javascript

```
return myVariable;
```

We could create a function to create a greeting message and return the value back to the caller

javascript

```
function createGreetingMessage(name) {  
  const message = `Hello, ${name}`;  
  return message;  
}
```

When calling this function we'll store the value in a variable. This is much the same way we'd set a variable to a static value (like `const name = 'Christopher'`).

javascript

```
const greetingMessage = createGreetingMessage('Christopher');
```

Functions as parameters for functions

As you progress in your programming career, you will come across functions which accept functions as parameters. This neat trick is commonly used when we don't know when something is going to occur or complete, but we know we need to perform an operation in response.

As an example, consider `setTimeout`, which begins a timer and will execute code when it completes. We need to tell it what code we want to execute. Sounds like a perfect job for a function!

If you run the code below, after 3 seconds you'll see the message **3 seconds has elapsed**.

javascript

```
function displayDone() {
  console.log('3 seconds has elapsed');
}
// timer value is in milliseconds
setTimeout(displayDone, 3000);
```

Anonymous functions

Let's take another look at what we've built. We're creating a function with a name which will be used one time. As our application gets more complex, we can see ourselves creating a lot of functions which will only be called once. This isn't ideal. As it turns out, we don't always need to provide a name!

When we are passing a function as a parameter we can bypass creating one in advance and instead build one as part of the parameter. We use the same `function` keyword, but instead we build it as a parameter.

Let's rewrite the code above to use an anonymous function:

javascript

```
setTimeout(function() {
  console.log('3 seconds has elapsed');
}, 3000);
```

If you run our new code you'll notice we get the same results. We've created a function, but didn't have to give it a name!

Fat arrow functions

One shortcut common in a lot of programming languages (including JavaScript) is the ability to use what's called an **arrow** or **fat arrow** function. It uses a special indicator of `=>`, which looks like an arrow - thus the name! By using `=>`, we are able to skip the `function` keyword.

Let's rewrite our code one more time to use a fat arrow function:

javascript

```
setTimeout(() => {  
  console.log('3 seconds has elapsed');  
}, 3000);
```

When to use each strategy

You've now seen we have three ways to pass a function as a parameter and might be wondering when to use each. If you know you'll be using the function more than once, create it as normal. If you'll be using it for just the one location, it's generally best to use an anonymous function. Whether or not you use a fat arrow function or the more traditional `function` syntax is up to you, but you will notice most modern developers prefer `=>`.

Challenge

Can you articulate in one sentence the difference between functions and methods? Give it a try!

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

It's worth [reading up a little more on arrow functions](#), as they are increasingly used in code bases. Practice writing a function, and then rewriting it with this syntax.

Assignment

JavaScript Basics: Making Decisions

JavaScript Basics Making Decisions

Booleans `true` `false`

```
let myStatement = true  
let anotherStatement = false
```

If statements

```
if (Status === 200) {  
  message = 'OK';  
} else {  
  message = 'Error!';  
}
```

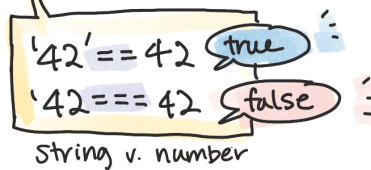
Condition

↓ Ternary

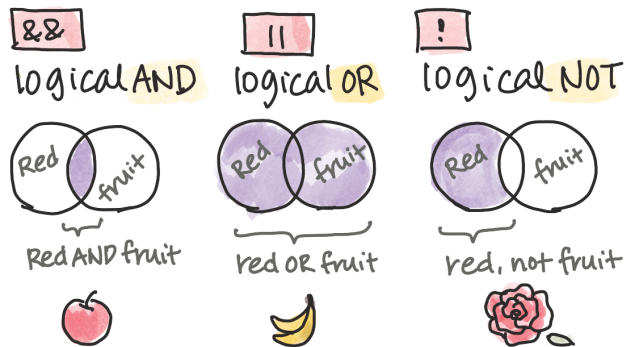
```
const message =  
(Status === 200) ? 'OK' : 'Error!';
```

Comparing values

- `>` Greater than
- `<` Less than
- `>=` Greater than or equal to
- `<=` Less than or equal to
- `==` equal values
- `!=` not equal values
- `===` equal values + data types
- `!==` not equal values + data types



Logical Operators



@AzureAdvocates / @ginie_mac

Sketchnote by Tomomi Imura

Pre-Lecture Quiz

Pre-lecture quiz

Making decisions and controlling the order in which your code runs makes your code reusable and robust. This section covers the syntax for controlling data flow in JavaScript and its significance when used with Boolean data types



 Click the image above for a video about making decisions.

You can take this lesson on [Microsoft Learn!](#)

A Brief Recap on Booleans

Booleans can be only two values: `true` or `false`. Booleans help make decisions on which lines of code should run when certain conditions are met.

Set your boolean to be true or false like this:

```
let myTrueBool = true  let myFalseBool = false
```

✅ Booleans are named after the English mathematician, philosopher and logician George Boole (1815–1864).

Comparison Operators and Booleans

Operators are used to evaluate conditions by making comparisons that will create a Boolean value. The following is a list of operators that are frequently used.

Symbol	Description	Example
<	Less than: Compares two values and returns the <code>true</code> Boolean data type if the value on the left side is less than the right	<code>5 < 6 // true</code>
<=	Less than or equal to: Compares two values and returns the <code>true</code> Boolean data type if the value on the left side is less than or equal to the right	<code>5 <= 6 // true</code>
>	Greater than: Compares two values and returns the <code>true</code> Boolean data type if the value on the left side is larger than the right	<code>5 > 6 // false</code>
>=	Greater than or equal to: Compares two values and returns the <code>true</code> Boolean data type if the value on the left side is larger than or equal to the right	<code>5 >= 6 // false</code>
===	Strict equality: Compares two values and returns the <code>true</code> Boolean data type if values on the right and left are equal AND are the same data type.	<code>5 === 6 // false</code>
!==	Inequality: Compares two values and returns the opposite Boolean value of what a strict equality operator would return	<code>5 !== 6 // true</code>

✔ Check your knowledge by writing some comparisons in your browser's console. Does any returned data surprise you?

If Statement

The if statement will run code in between its blocks if the condition is true.

javascript

```
if (condition){
  //Condition was true. Code in this block will run.
}
```

Logical operators are often used to form the condition.

```
let currentMoney;
let laptopPrice;

if (currentMoney >= laptopPrice){
  //Condition was true. Code in this block will run.
  console.log("Getting a new laptop!");
}
```

IF..Else Statement

The `else` statement will run the code in between its blocks when the condition is false. It's optional with an `if` statement.

```
let currentMoney;
let laptopPrice;

if (currentMoney >= laptopPrice){
  //Condition was true. Code in this block will run.
  console.log("Getting a new laptop!");
}
else{
  //Condition was false. Code in this block will run.
  console.log("Can't afford a new laptop, yet!");
}
```

✅ Test your understanding of this code and the following code by running it in a browser console. Change the values of the `currentMoney` and `laptopPrice` variables to change the returned `console.log()` .

Logical Operators and Booleans

Decisions might require more than one comparison, and can be strung together with logical operators to produce a Boolean value.

Symbol	Description	Example
--------	-------------	---------

Symbol	Description	Example
	Logical AND: Compares two Boolean expressions.	
&&	Returns true only if both sides are true	<code>(5 > 6) && (5 < 6) //One side is false, other is true.</code>
	Logical OR: Compares two Boolean expressions.	
	Returns true if at least one side is true	<code>(5 > 6) (5 < 6) //One side is false, other is true. R</code>
	Logical NOT: Returns the opposite value of a Boolean expression	
!		<code>!(5 > 6) // 5 is not greater than 6, but "!" will return</code>

Conditions and Decisions with Logical Operators

Logical operators can be used to form conditions in if..else statements.

javascript

```
let currentMoney;
let laptopPrice;
let laptopDiscountPrice = laptopPrice - (laptopPrice * .20) //Laptop price

if (currentMoney >= laptopPrice || currentMoney >= laptopDiscountPrice){
```

```
    //Condition was true. Code in this block will run.
    console.log("Getting a new laptop!");
}
else {
    //Condition was true. Code in this block will run.
    console.log("Can't afford a new laptop, yet!");
}
```

Negation operator

You've seen so far how if you can use an `if...else` statement to create conditional logic. Anything that goes into an `if` needs to evaluate to true/false. By using the `!` operator you can *negate* the expression. It would look like so:

javascript

```
if (!condition) {
    // runs if condition is false
} else {
    // runs if condition is true
}
```

Ternary expressions

`if...else` isn't the only way to express decision logic. You can also use something called a ternary operator. The syntax for it looks like this:

javascript

```
let variable = condition ? <return this if true> : <return this if false>
```

Below is a more tangible example:

javascript

```
let firstNumber = 20;
let secondNumber = 10
let biggestNumber = firstNumber > secondNumber ? firstNumber : secondNumber;
```

✅ Take a minute to read this code a few times. Do you understand how these operators are working?

The above states that

- if `firstNumber` is larger than `secondNumber`
- then assign `firstNumber` to `biggestNumber`
- else assign `secondNumber` .

The ternary expression is just a compact way of writing the code below:

javascript

```
let biggestNumber;
if (firstNumber > secondNumber) {
  biggestNumber = firstNumber;
} else {
  biggestNumber = secondNumber;
}
```

Challenge

Create a program that is written first with logical operators, and then rewrite it using a ternary expression. What's your preferred syntax?

Post-Lecture Quiz

[Post-lecture quiz](#)

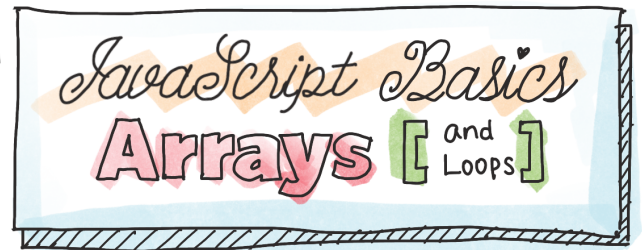
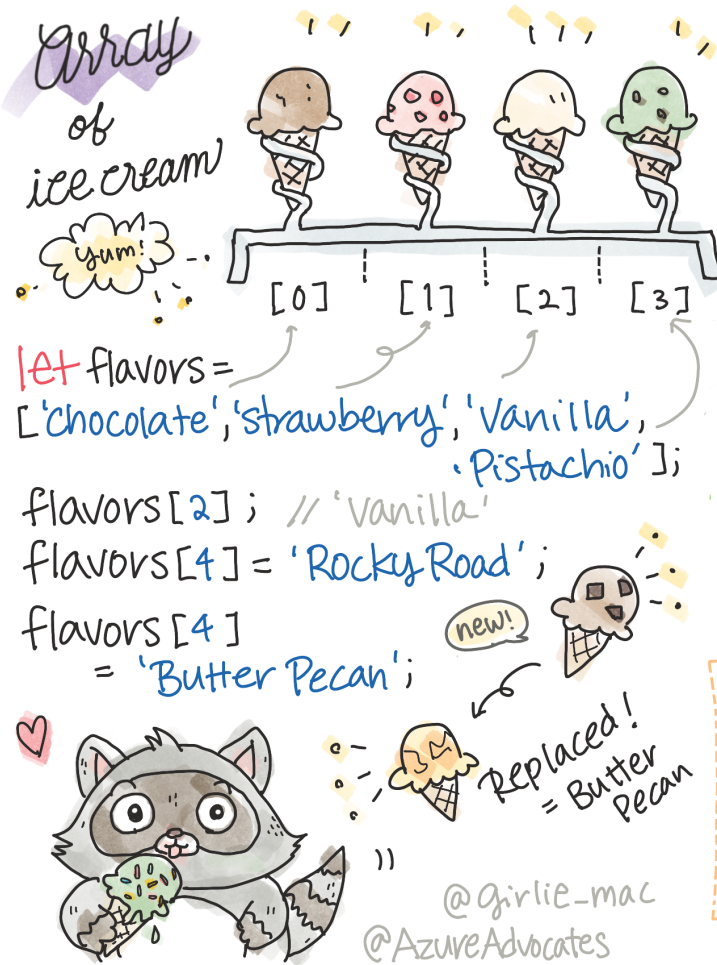
Review & Self Study

Read more about the many operators available to the user [on MDN](#).

Go through Josh Comeau's wonderful [operator lookup!](#)

Assignment

JavaScript Basics: Arrays and Loops



For Loop

```
for (let i = 0; i < flavors.length; i++) {
  console.log(flavors[i]);
}
```

prints out each flavor after each iteration!

While-Loop

```
let i = 0;
while (i < flavors.length) {
  console.log(flavors[i]);
  i++;
}
```

The loop will stop when the condition is met!

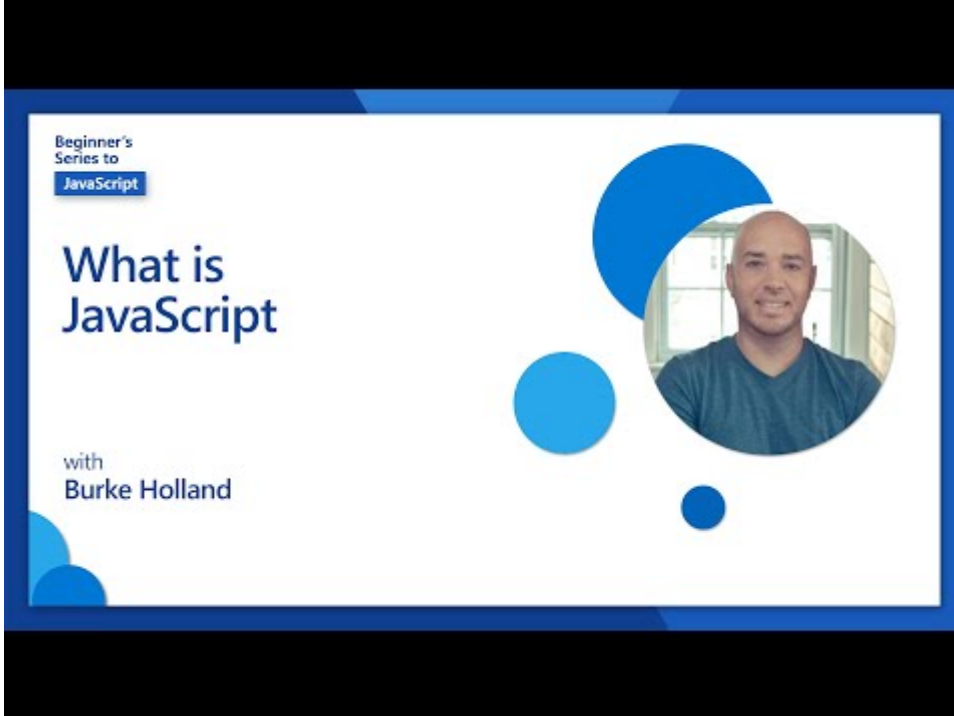
Loops

Sketchnote by [Tomomi Imura](#)

Pre-Lecture Quiz

Pre-lecture quiz

This lesson covers the basics of JavaScript, the language that provides interactivity on the web. In this lesson, you'll learn about arrays and loops, which are used to manipulate data.



📺 Click the image above for a video about arrays and loops.

You can take this lesson on [Microsoft Learn!](#)

Arrays

Working with data is a common task for any language, and it's a much easier task when data is organized in a structural format, such as arrays. With arrays, data is stored in a structure similar to a list. One major benefit of arrays is that you can store different types of data in one array.

✅ Arrays are all around us! Can you think of a real-life example of an array, such as a solar panel array?

The syntax for an array is a pair of square brackets.

```
let myArray = [];
```

This is an empty array, but arrays can be declared already populated with data. Multiple values in an array are separated by a comma.

```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla", "Pistachio", "Rocky"];
```

The array values are assigned a unique value called the **index**, a whole number that is assigned based on its distance from the beginning of the array. In the example above, the string value "Chocolate" has an index of 0, and the index of "Rocky Road" is 4. Use the index with square brackets to retrieve, change, or insert array values.

✅ Does it surprise you that arrays start at the zero index? In some programming languages, indexes start at 1. There's an interesting history around this, which you can [read on Wikipedia](#).

javascript

```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla", "Pistachio", '
iceCreamFlavors[2]; // "Vanilla"
```

You can leverage the index to change a value, like this:

javascript

```
iceCreamFlavors[4] = "Butter Pecan"; // Changed "Rocky Road" to "Butter Pecan"
```

And you can insert a new value at a given index like this:

javascript

```
iceCreamFlavors[5] = "Cookie Dough"; // Added "Cookie Dough"
```

✅ A more common way to push values to an array is by using array operators such as `array.push()`

To find out how many items are in an array, use the `length` property.

javascript

```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla", "Pistachio", '
iceCreamFlavors.length; // 5
```

✅ Try it yourself! Use your browser's console to create and manipulate an array of your own creation.

Loops

Loops allow for repetitive or **iterative** tasks, and can save a lot of time and code. Each iteration can vary in their variables, values, and conditions. There are different types of loops in JavaScript, and they have small differences, but essentially do the same thing: loop over data.

For Loop

The `for` loop requires 3 parts to iterate: - `counter` A variable that is typically initialized with a number that counts the number of iterations. - `condition` Expression that uses comparison operators to cause the loop to stop when `true` - `iteration-expression` Runs at the end of each iteration, typically used to change the counter value

javascript

```
//Counting up to 10
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

✅ Run this code in a browser console. What happens when you make small changes to the counter, condition, or iteration expression? Can you make it run backwards, creating a countdown?

While loop

Unlike the syntax for the `for` loop, `while` loops only require a condition that will stop the loop when `true`. Conditions in loops usually rely on other values like counters, and must be managed during the loop. Starting values for counters must be created outside the loop, and any expressions to meet a condition, including changing the counter must be maintained inside the loop.

javascript

```
//Counting up to 10
let i = 0;
while (i < 10) {
  console.log(i);
  i++;
}
```

✅ Why would you choose a for loop vs. a while loop? 17K viewers had the same question on StackOverflow, and some of the opinions [might be interesting to you](#).

Loops and Arrays

Arrays are often used with loops because most conditions require the length of the array to stop the loop, and the index can also be the counter value.

javascript

```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla", "Pistachio", ']
```

```
for (let i = 0; i < iceCreamFlavors.length; i++) {  
  console.log(iceCreamFlavors[i]);  
} //Ends when all flavors are printed
```

- ✅ Experiment with looping over an array of your own making in your browser's console.
-

Challenge

There are other ways of looping over arrays other than for and while loops. There are [forEach](#), [for-of](#), and [map](#). Rewrite your array loop using one of these techniques.

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Arrays in JavaScript have many methods attached to them, extremely useful for data manipulation. [Read up on these methods](#) and try some of them out (like push, pop, slice and splice) on an array of your creation.

Assignment

[Loop an Array](#)

Terrarium Project Part 1: Introduction to HTML

Introduction to HTML

HTML "skelton" { CSS to dress up, JS brings life }

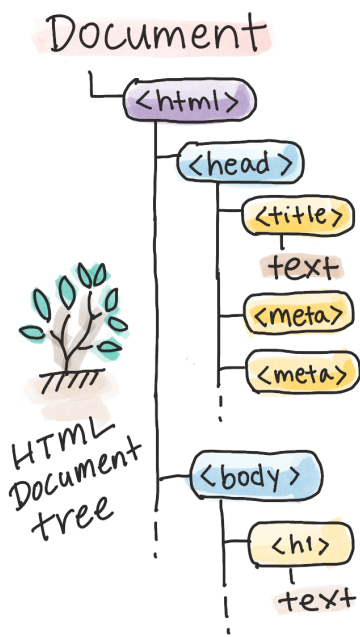

Semantic Markup

HTML tags // represent data // not defining how it look! // SEO

Screen reader

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>welcome!</title>  
    <meta charset="utf-8">  
    <meta http-equiv="X-UA-Compatible" ...>  
    <meta name="viewport" ...>  
  </head>  
  <body>  
    <h1>my Terrarium</h1>  
    <div id="page">  
        
      ...  
    </div>  
  </body>  
</html>
```

VSCode, index.html, open!



@AzureAdvocates
@girlie-mac

Sketchnote by [Tomomi Imura](#)

Pre-Lecture Quiz

[Pre-lecture quiz](#)

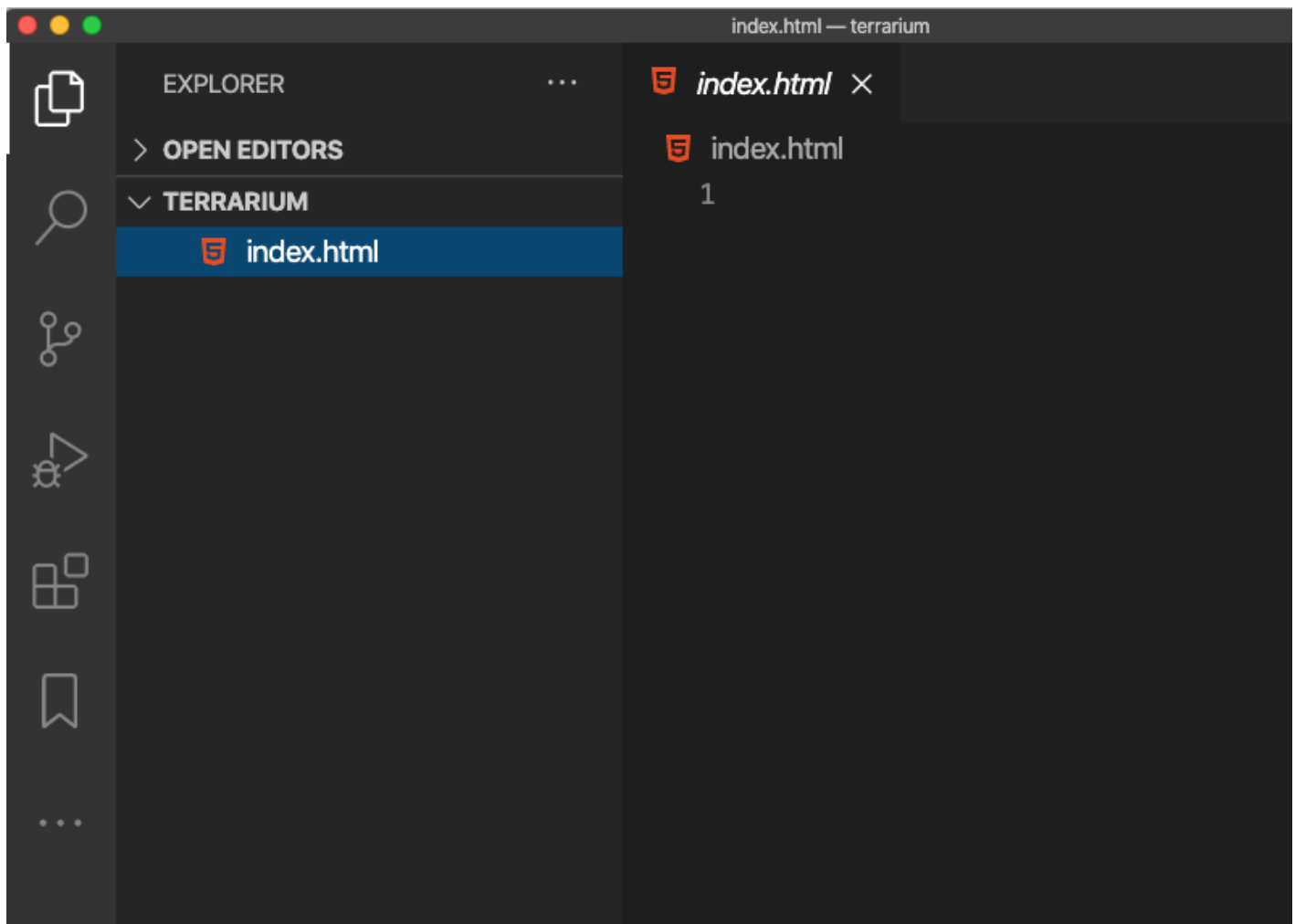
Introduction

HTML, or HyperText Markup Language, is the 'skeleton' of the web. If CSS 'dresses up' your HTML and JavaScript brings it to life, HTML is the body of your web application. HTML's syntax even reflects that idea, as it includes "head", "body", and "footer" tags.

In this lesson, we're going to use HTML to layout the 'skeleton' of our virtual terrarium's interface. It will have a title and three columns: a right and a left column where the draggable plants live, and a center area that will be the actual glass-looking terrarium. By the end of this lesson, you will be able to see the plants in the columns, but the interface will look a little strange; don't worry, in the next section you will add CSS styles to the interface to make it look better.

Task

On your computer, create a folder called 'terrarium' and inside it, a file called 'index.html'. You can do this in Visual Studio Code after you create your terrarium folder by opening a new VS Code window, clicking 'open folder', and navigating to your new folder. Click the small 'file' button in the Explorer pane and create the new file:



Or

Use these commands on your git bash:

- `mkdir terrarium`
- `cd terrarium`
- `touch index.html`
- `code index.html` or `nano index.html`

index.html files indicate to a browser that it is the default file in a folder; URLs such as `https://anysite.com/test` might be built using a folder structure including a folder called `test` with `index.html` inside it; `index.html` doesn't have to show in a URL.

The DocType and html tags

The first line of an HTML file is its doctype. It's a little surprising that you need to have this line at the very top of the file, but it tells older browsers that the browser needs to render the page in a standard mode, following the current html specification.

Tip: in VS Code, you can hover over a tag and get information about its use from the MDN Reference guides.

The second line should be the `<html>` tag's opening tag, followed right now by its closing tag `</html>`. These tags are the root elements of your interface.

Task

Add these lines at the top of your `index.html` file:

HTML

```
<!DOCTYPE html>
<html></html>
```

✓ There are a few different modes that can be determined by setting the DocType with a query string: [Quirks Mode and Standards Mode](#). These modes used to support really old browsers that aren't normally used nowadays (Netscape Navigator 4 and Internet Explorer 5). You can stick to the standard doctype declaration.

The document's 'head'

The 'head' area of the HTML document includes crucial information about your web page, also known as [metadata](#). In our case, we tell the web server to which this page will be sent to be rendered, these four things:

- the page's title
- page metadata including:
 - the 'character set', telling about what character encoding is used in the page

- browser information, including `x-ua-compatible` which indicates that the IE=edge browser is supported
- information about how the viewport should behave when it is loaded. Setting the viewport to have an initial scale of 1 controls the zoom level when the page is first loaded.

Task

Add a 'head' block to your document in between the opening and closing `<html>` tags.

html

```
<head>
  <title>Welcome to my Virtual Terrarium</title>
  <meta charset="utf-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
</head>
```

✓ What would happen if you set a viewport meta tag like this:

`<meta name="viewport" content="width=600">` ? Read more about the [viewport](#).

The document's body

HTML Tags

In HTML, you add tags to your .html file to create elements of a web page. Each tag usually has an opening and closing tag, like this: `<p>hello</p>` to indicate a paragraph. Create your interface's body by adding a set of `<body>` tags inside the `<html>` tag pair; your markup now looks like this:

Task

html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to my Virtual Terrarium</title>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1'
  </head>
  <body></body>
</html>
```

Now, you can start building out your page. Normally, you use `<div>` tags to create the separate elements in a page. We'll create a series of `<div>` elements which will contain images.

Images

One html tag that doesn't need a closing tag is the `` tag, because it has a `src` element that contains all the information the page needs to render the item.

Create a folder in your app called `images` and in that, add all the images in the [source code folder](#); (there are 14 images of plants).

Task

Add those plant images into two columns between the `<body></body>` tags:

html

```
<div id="page">
  <div id="left-container" class="container">
    <div class="plant-holder">
      
    </div>
    <div class="plant-holder">
      
    </div>
    <div class="plant-holder">
      
    </div>
    <div class="plant-holder">
      
    </div>
    <div class="plant-holder">
      
    </div>
    <div class="plant-holder">
      
    </div>
    <div class="plant-holder">
      
    </div>
  </div>
</div>
```

```
    </div>
</div>
<div id="right-container" class="container">
  <div class="plant-holder">
    
  </div>
  <div class="plant-holder">
    
  </div>
  <div class="plant-holder">
    
  </div>
  <div class="plant-holder">
    
  </div>
  <div class="plant-holder">
    
  </div>
  <div class="plant-holder">
    
  </div>
  <div class="plant-holder">
    
  </div>
</div>
</div>
```

Note: Spans vs. Divs. Divs are considered 'block' elements, and Spans are 'inline'. What would happen if you transformed these divs to spans?

With this markup, the plants now show up on the screen. It looks pretty bad, because they aren't yet styled using CSS, and we'll do that in the next lesson.

Each image has alt text that will appear even if you can't see or render an image. This is an important attribute to include for accessibility. Learn more about accessibility in future lessons; for now, remember that the alt attribute provides alternative information for an image if a user for some reason cannot view it (because of slow connection, an error in the src attribute, or if the user uses a screen reader).

✅ Did you notice that each image has the same alt tag? Is this good practice? Why or why not? Can you improve this code?

Semantic markup

In general, it's preferable to use meaningful 'semantics' when writing HTML. What does that mean? It means that you use HTML tags to represent the type of data or interaction they were designed for. For example, the main title text on a page should use an `<h1>` tag.

Add the following line right below your opening `<body>` tag:

html

```
<h1>My Terrarium</h1>
```

Using semantic markup such as having headers be `<h1>` and unordered lists be rendered as `` helps screen readers navigate through a page. In general, buttons should be written as `<button>` and lists should be ``. While it's *possible* to use specially styled `` elements with click handlers to mock buttons, it's better for disabled users to use technologies to determine where on a page a button resides, and to interact with it, if the element appears as a button. For this reason, try to use semantic markup as much as possible.

✅ Take a look at a screen reader and [how it interacts with a web page](#). Can you see why having non semantic markup might frustrate the user?

The terrarium

The last part of this interface involves creating markup that will be styled to create a terrarium.

Task:

Add this markup above the last `</div>` tag:

html

```
<div id="terrarium">
  <div class="jar-top"></div>
  <div class="jar-walls">
    <div class="jar-glossy-long"></div>
    <div class="jar-glossy-short"></div>
  </div>
  <div class="dirt"></div>
```

```
<div class="jar-bottom"></div>  
</div>
```

✔ Even though you added this markup to the screen, you see absolutely nothing render. Why?

Challenge

There are some wild 'older' tags in HTML that are still fun to play with, though you shouldn't use deprecated tags such as [these tags](#) in your markup. Still, can you use the old `<marquee>` tag to make the h1 title scroll horizontally? (if you do, don't forget to remove it afterwards)

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

HTML is the 'tried and true' building block system that has helped build the web into what it is today. Learn a little about its history by studying some old and new tags. Can you figure out why some tags were deprecated and some added? What tags might be introduced in the future?

Learn more about building sites for the web and mobile devices at [Microsoft Learn](#).

Assignment

[Practice your HTML: Build a blog mockup](#)

Terrarium Project Part 2: Introduction to CSS

Introduction to CSS

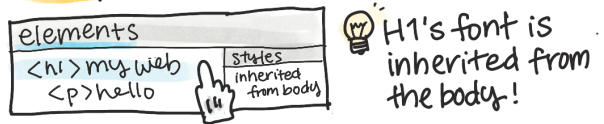


Inheritance
 body {
 font-family: helvetica,
 arial, sans-serif; }

Cascade

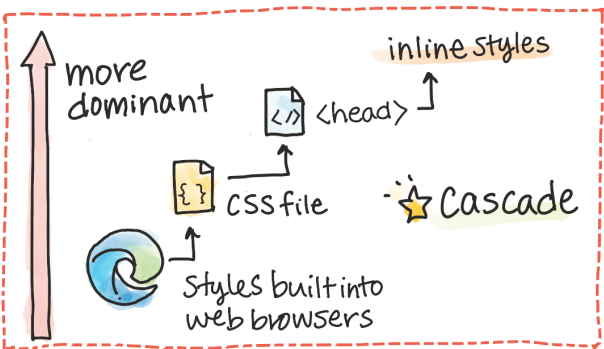
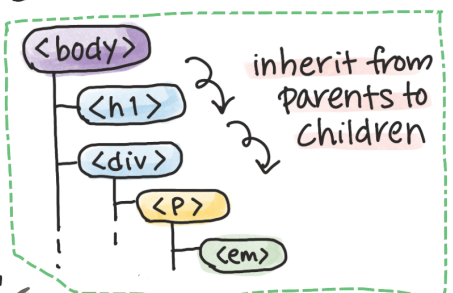


Inspect in browser dev tools



Selectors

- ♥ tag
- ♥ id
- ♥ class



```
<div id="plant1" class="plant"> #plant1 { ... }
```

Layouts

- ♥ positioning
- ♥ display rules
- ♥ flows etc.

```
.plant {  
  position: absolute;  
  width: 150%;  
  z-index: 2;  
}
```

@AzureAdvocates / @girlie-mac

Sketchnote by Tomomi Imura

Pre-Lecture Quiz

Pre-lecture quiz

Introduction

CSS, or Cascading Style Sheets, solve an important problem of web development: how to make your web site look nice. Styling your apps makes them more usable and nicer-looking; you can also use CSS to create Responsive Web Design (RWD) - allowing your apps to look good no matter what screen size they are displayed on. CSS is not only about making your app look nice; its spec includes animations and transforms that can enable sophisticated interactions for your apps. The CSS Working Group helps maintain current CSS specifications; you can follow their work at World Wide Web Consortium's site.

Note, CSS is a language that evolves, like everything on the web, and not all browsers support newer parts of the specification. Always check your implementations by consulting [CanIUse.com](https://caniuse.com).

In this lesson, we're going to add styles to our online terrarium and learn more about several CSS concepts: the cascade, inheritance, and the use of selectors, positioning, and using CSS to build layouts. In the process we will layout the terrarium and create the actual terrarium itself.

Prerequisite

You should have the HTML for your terrarium built and ready to be styled.

Task

In your terrarium folder, create a new file called `style.css`. Import that file in the `<head>` section:

html

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

The Cascade

Cascading Style Sheets incorporate the idea that the styles 'cascade' such that the application of a style is guided by its priority. Styles set by a web site author take priority over those set by a browser. Styles set 'inline' take priority over those set in an external style sheet.

Task

Add the inline style "color: red" to your `<h1>` tag:

HTML

```
<h1 style="color: red">My Terrarium</h1>
```

Then, add the following code to your `style.css` file:

```
h1 {  
  color: blue;  
}
```

- ✓ Which color displays in your web app? Why? Can you find a way to override styles? When would you want to do this, or why not?
-

Inheritance

Styles are inherited from an ancestor style to a descendent, such that nested elements inherit the styles of their parents.

Task

Set the body's font to a given font, and check to see a nested element's font:

```
body {  
  font-family: helvetica, arial, sans-serif;  
}
```

CSS

Open your browser's console to the 'Elements' tab and observe the H1's font. It inherits its font from the body, as stated within the browser:

A screenshot of a browser's developer console. At the top, a grey box says "Inherited from body". Below it, the CSS rule for the body is shown: "body { font-family: helvetica, arial, sans-serif; }". The source is identified as "style.css:1".

```
Inherited from body  
body {  
  font-family: helvetica, arial, sans-serif;  
} style.css:1
```

- ✓ Can you make a nested style inherit a different property?
-

CSS Selectors

Tags

So far, your `style.css` file has only a few tags styled, and the app looks pretty strange:

CSS

```
body {
  font-family: helvetica, arial, sans-serif;
}

h1 {
  color: #3a241d;
  text-align: center;
}
```

This way of styling a tag gives you control over unique elements, but you need to control the styles of many plants in your terrarium. To do that, you need to leverage CSS selectors.

Ids

Add some style to layout the left and right containers. Since there is only one left container and only one right container, they are given ids in the markup. To style them, use `#` :

CSS

```
#left-container {
  background-color: #eee;
  width: 15%;
  left: 0px;
  top: 0px;
  position: absolute;
  height: 100%;
  padding: 10px;
}

#right-container {
  background-color: #eee;
  width: 15%;
  right: 0px;
  top: 0px;
  position: absolute;
  height: 100%;
  padding: 10px;
}
```

Here, you have placed these containers with absolute positioning to the far left and right of the screen, and used percentages for their width so that they can scale for small mobile screens.

✔ This code is quite repeated, thus not "DRY" (Don't Repeat Yourself); can you find a better way to style these ids, perhaps with an id and a class? You would need to change the markup and refactor the CSS:

html

```
<div id="left-container" class="container"></div>
```

Classes

In the example above, you styled two unique elements on the screen. If you want styles to apply to many elements on the screen, you can use CSS classes. Do this to layout the plants in the left and right containers.

Notice that each plant in the HTML markup has a combination of ids and classes. The ids here are used by the JavaScript that you will add later to manipulate the terrarium plant placement. The classes, however, give all the plants a given style.

html

```
<div class="plant-holder">
  
</div>
```

Add the following to your `style.css` file:

CSS

```
.plant-holder {
  position: relative;
  height: 13%;
  left: -10px;
}

.plant {
  position: absolute;
  max-width: 150%;
  max-height: 150%;
  z-index: 2;
}
```

Notable in this snippet is the mixture of relative and absolute positioning, which we'll cover in the next section. Take a look at the way heights are handled by percentages:

You set the height of the plant holder to 13%, a good number to ensure that all the plants are displayed in each vertical container without need for scrolling.

You set the plant holder to move to the left to allow the plants to be more centered within their container. The images have a large amount of transparent background so as to make them more draggable, so need to be pushed to the left to fit better on the screen.

Then, the plant itself is given a max-width of 150%. This allows it to scale down as the browser scales down. Try resizing your browser; the plants stay in their containers but scale down to fit.

Also notable is the use of z-index, which controls the relative altitude of an element (so that the plants sit on top of the container and appear to sit inside the terrarium).

✅ Why do you need both a plant holder and a plant CSS selector?

CSS Positioning

Mixing position properties (there are static, relative, fixed, absolute, and sticky positions) can be a little tricky, but when done properly it gives you good control over the elements on your pages.

Absolute positioned elements are positioned relative to their nearest positioned ancestors, and if there are none, it is positioned according to the document body.

Relative positioned elements are positioned based on the CSS's directions to adjust its placement away from its initial position.

In our sample, the `plant-holder` is a relative-positioned element that is positioned within an absolute-positioned container. The resultant behavior is that the side bar containers are pinned left and right, and the plant-holder is nested, adjusting itself within the side bars, giving space for the plants to be placed in a vertical row.

The `plant` itself also has absolute positioning, necessary to making it draggable, as you will discover in the next lesson.

✅ Experiment with switching the types of positioning of the side containers and the plant-holder. What happens?

CSS Layouts

Now you will use what you learned to build the terrarium itself, all using CSS!

First, style the `.terrarium` div children as a rounded rectangle using CSS:

CSS

```
.jar-walls {
  height: 80%;
  width: 60%;
  background: #d1e1df;
  border-radius: 1rem;
  position: absolute;
  bottom: 0.5%;
  left: 20%;
  opacity: 0.5;
  z-index: 1;
}

.jar-top {
  width: 50%;
  height: 5%;
  background: #d1e1df;
  position: absolute;
  bottom: 80.5%;
  left: 25%;
  opacity: 0.7;
  z-index: 1;
}

.jar-bottom {
  width: 50%;
  height: 1%;
  background: #d1e1df;
  position: absolute;
  bottom: 0%;
  left: 25%;
  opacity: 0.7;
}

.dirt {
  width: 60%;
  height: 5%;
  background: #3a241d;
  position: absolute;
  border-radius: 0 0 1rem 1rem;
  bottom: 1%;
}
```

```
    left: 20%;
    opacity: 0.7;
    z-index: -1;
}
```

Note the use of percentages here. If you scale your browser down, you can see how the jar scales as well. Also notice the widths and height percentages for the jar elements and how each element is absolutely positioned in the center, pinned to the bottom of the viewport.

We are also using `rem` for the border-radius, a font-relative length. Read more about this type of relative measurement in the [CSS spec](#).

✅ Try changing the jar colors and opacity vs. those of the dirt. What happens? Why?

Challenge

Add a 'bubble' shine to the left bottom area of the jar to make it look more glasslike. You will be styling the `.jar-glossy-long` and `.jar-glossy-short` to look like a reflected shine. Here's how it would look:

My Terrarium



To complete the post-lecture quiz, go through this Learn module: [Style your HTML app with CSS](#)

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

CSS seems deceptively straightforward, but there are many challenges when trying to style an app perfectly for all browsers and all screen sizes. CSS-Grid and Flexbox are tools that have been developed to make the job a little more structured and more reliable. Learn about these tools by playing [Flexbox Froggy](#) and [Grid Garden](#).

Assignment

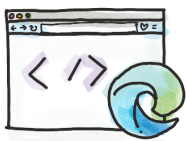
CSS Refactoring

Terrarium Project Part 3: DOM

Manipulation and a Closure

JavaScript
DOM & Closure
manipulation
Document Object Model

@AzureAdvocates
@girlie_mac



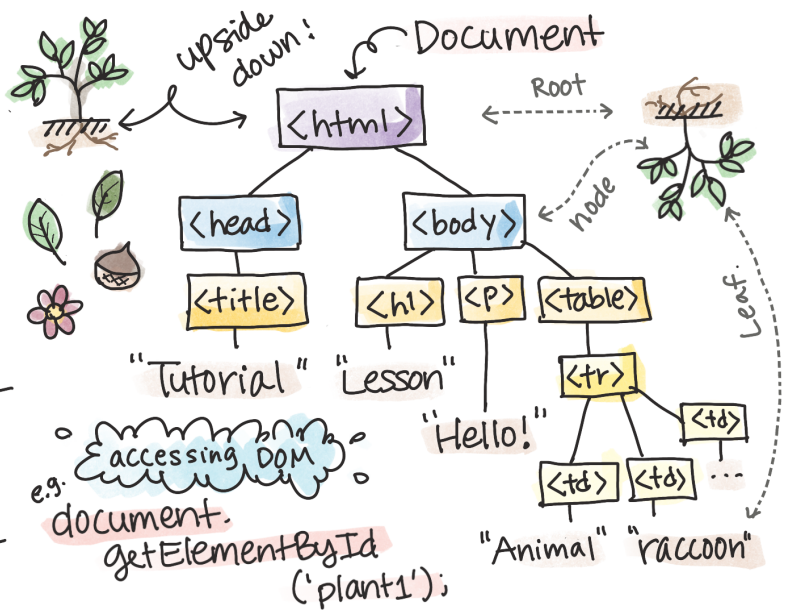
- ♥ programming interface for HTML
- ♥ data representation of the objects that make up the web structure + content

Closure

↳ an outer function that encloses an inner function



★ A closure gives you access to an outer function's scope from an inner function!



```
function dragElement(terrariumEl) {  
  let pos1=0, pos2=0, pos3=0;  
  terrariumEl.onpointerdown = pointerDrag;  
  
  function pointerDrag(e) {  
    e.preventDefault();  
    pos3 = e.clientX;  
    ...  
  }  
}
```

Sketchnote by Tomomi Imura

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

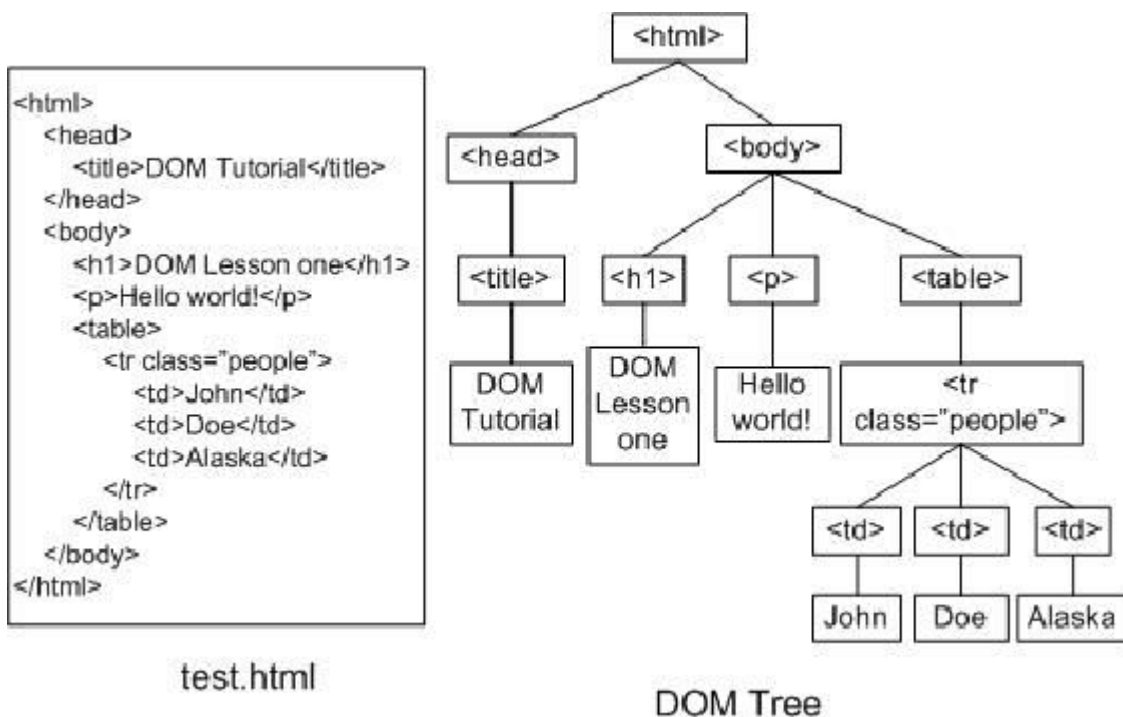
Manipulating the DOM, or the "Document Object Model", is a key aspect of web development. According to [MDN](#), "The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web." The challenges around DOM manipulation on the web have often been the impetus behind using JavaScript frameworks instead of vanilla JavaScript to manage the DOM, but we will manage on our own!

In addition, this lesson will introduce the idea of a [JavaScript closure](#), which you can think of as a function enclosed by another function so that the inner function has access to the outer function's scope.

JavaScript closures are a vast and complex topic. This lesson touches on the most basic idea that in this terrarium's code, you will find a closure: an inner function and an outer function constructed in a way to allow the inner function access to the outer function's scope. For much more information on how this works, please visit the [extensive documentation](#).

We will use a closure to manipulate the DOM.

Think of the DOM as a tree, representing all the ways that a web page document can be manipulated. Various APIs (Application Program Interfaces) have been written so that programmers, using their programming language of choice, can access the DOM and edit, change, rearrange, and otherwise manage it.



In this lesson, we will complete our interactive terrarium project by creating the JavaScript that will allow a user to manipulate the plants on the page.

Prerequisite

You should have the HTML and CSS for your terrarium built. By the end of this lesson you will be able to move the plants into and out of the terrarium by dragging them.

Task

In your terrarium folder, create a new file called `script.js`. Import that file in the `<head>` section:

```
<script src="./script.js" defer></script>
```

html

Note: use `defer` when importing an external JavaScript file into the html file so as to allow the JavaScript to execute only after the HTML file has been fully loaded. You could also use the `async` attribute, which allows the script to execute while the HTML file is parsing, but in our case, it's important to have the HTML elements fully available for dragging before we allow the drag script to be executed.

The DOM elements

The first thing you need to do is to create references to the elements that you want to manipulate in the DOM. In our case, they are the 14 plants currently waiting in the side bars.

Task

```
dragElement(document.getElementById('plant1'));  
dragElement(document.getElementById('plant2'));  
dragElement(document.getElementById('plant3'));
```

html

```
dragElement(document.getElementById('plant4'));
dragElement(document.getElementById('plant5'));
dragElement(document.getElementById('plant6'));
dragElement(document.getElementById('plant7'));
dragElement(document.getElementById('plant8'));
dragElement(document.getElementById('plant9'));
dragElement(document.getElementById('plant10'));
dragElement(document.getElementById('plant11'));
dragElement(document.getElementById('plant12'));
dragElement(document.getElementById('plant13'));
dragElement(document.getElementById('plant14'));
```

What's going on here? You are referencing the document and looking through its DOM to find an element with a particular Id. Remember in the first lesson on HTML that you gave individual Ids to each plant image (`id="plant1"`)? Now you will make use of that effort. After identifying each element, you pass that item to a function called `dragElement` that you'll build in a minute. Thus, the element in the HTML is now drag-enabled, or will be shortly.

✅ Why do we reference elements by Id? Why not by their CSS class? You might refer to the previous lesson on CSS to answer this question.

The Closure

Now you are ready to create the `dragElement` closure, which is an outer function that encloses an inner function or functions (in our case, we will have three).

Closures are useful when one or more functions need to access an outer function's scope. Here's an example:

javascript

```
function displayCandy(){
  let candy = ['jellybeans'];
  function addCandy(candyType) {
    candy.push(candyType)
  }
  addCandy('gumdrops');
}
displayCandy();
console.log(candy)
```

In this example, the `displayCandy` function surrounds a function that pushes a new candy type into an array that already exists in the function. If you were to run this code, the `candy` array would be undefined, as it is a local variable (local to the closure).

✅ How can you make the `candy` array accessible? Try moving it outside the closure. This way, the array becomes global, rather than remaining only available to the closure's local scope.

Task

Under the element declarations in `script.js`, create a function:

javascript

```
function dragElement(terrariumElement) {
  //set 4 positions for positioning on the screen
  let pos1 = 0,
      pos2 = 0,
      pos3 = 0,
      pos4 = 0;
  terrariumElement.onpointerdown = pointerDrag;
}
```

`dragElement` gets its `terrariumElement` object from the declarations at the top of the script. Then, you set some local positions at `0` for the object passed into the function. These are the local variables that will be manipulated for each element as you add drag and drop functionality within the closure to each element. The terrarium will be populated by these dragged elements, so the application needs to keep track of where they are placed.

In addition, the `terrariumElement` that is passed to this function is assigned a `pointerdown` event, which is part of the [web APIs](#) designed to help with DOM management. `onpointerdown` fires when a button is pushed, or in our case, a draggable element is touched. This event handler works on both [web and mobile browsers](#), with a few exceptions.

✅ The [event handler](#) `onclick` has much more support cross-browser; why wouldn't you use it here? Think about the exact type of screen interaction you're trying to create here.

The Pointerdrag function

The `terrariumElement` is ready to be dragged around; when the `onpointerdown` event is fired, the function `pointerDrag` is invoked. Add that function right under this line:

```
terrariumElement.onpointerdown = pointerDrag; :
```

Task

javascript

```
function pointerDrag(e) {  
  e.preventDefault();  
  console.log(e);  
  pos3 = e.clientX;  
  pos4 = e.clientY;  
}
```

Several things happen. First, you prevent the default events that normally happen on pointerdown from occurring by using `e.preventDefault()`. This way you have more control over the interface's behavior.

Come back to this line when you've built the script file completely and try it without `e.preventDefault()` - what happens?

Second, open `index.html` in a browser window, and inspect the interface. When you click a plant, you can see how the 'e' event is captured. Dig into the event to see how much information is gathered by one pointer down event!

Next, note how the local variables `pos3` and `pos4` are set to `e.clientX`. You can find the `e` values in the inspection pane. These values capture the x and y coordinates of the plant at the moment you click on it or touch it. You will need fine-grained control over the behavior of the plants as you click and drag them, so you keep track of their coordinates.

✅ Is it becoming more clear why this entire app is built with one big closure? If it wasn't, how would you maintain scope for each of the 14 draggable plants?

Complete the initial function by adding two more pointer event manipulations under `pos4 = e.clientY` :

html

```
document.onpointermove = elementDrag;  
document.onpointerup = stopElementDrag;
```

Now you are indicating that you want the plant to be dragged along with the pointer as you move it, and for the dragging gesture to stop when you deselect the plant. `onpointermove` and

`onpointerup` are all parts of the same API as `onpointerdown`. The interface will throw errors now as you have not yet defined the `elementDrag` and the `stopElementDrag` functions, so build those out next.

The `elementDrag` and `stopElementDrag` functions

You will complete your closure by adding two more internal functions that will handle what happens when you drag a plant and stop dragging it. The behavior you want is that you can drag any plant at any time and place it anywhere on the screen. This interface is quite un-opinionated (there is no drop zone for example) to allow you to design your terrarium exactly as you like it by adding, removing, and repositioning plants.

Task

Add the `elementDrag` function right after the closing curly bracket of `pointerDrag`:

javascript

```
function elementDrag(e) {
  pos1 = pos3 - e.clientX;
  pos2 = pos4 - e.clientY;
  pos3 = e.clientX;
  pos4 = e.clientY;
  console.log(pos1, pos2, pos3, pos4);
  terrariumElement.style.top = terrariumElement.offsetTop - pos2 + 'px';
  terrariumElement.style.left = terrariumElement.offsetLeft - pos1 + 'px'
}
```

In this function, you do a lot of editing of the initial positions 1-4 that you set as local variables in the outer function. What's going on here?

As you drag, you reassign `pos1` by making it equal to `pos3` (which you set earlier as `e.clientX`) minus the current `e.clientX` value. You do a similar operation to `pos2`. Then, you reset `pos3` and `pos4` to the new X and Y coordinates of the element. You can watch these changes in the console as you drag. Then, you manipulate the plant's CSS style to set its new position based on the new positions of `pos1` and `pos2`, calculating the plant's top and left X and Y coordinates based on comparing its offset with these new positions.

`offsetTop` and `offsetLeft` are CSS properties that set an element's position based on that of its parent; its parent can be any element that is not positioned as `static`.

All this recalculation of positioning allows you to fine-tune the behavior of the terrarium and its plants.

Task

The final task to complete the interface is to add the `stopElementDrag` function after the closing curly bracket of `elementDrag` :

javascript

```
function stopElementDrag() {  
    document.onpointerup = null;  
    document.onpointermove = null;  
}
```

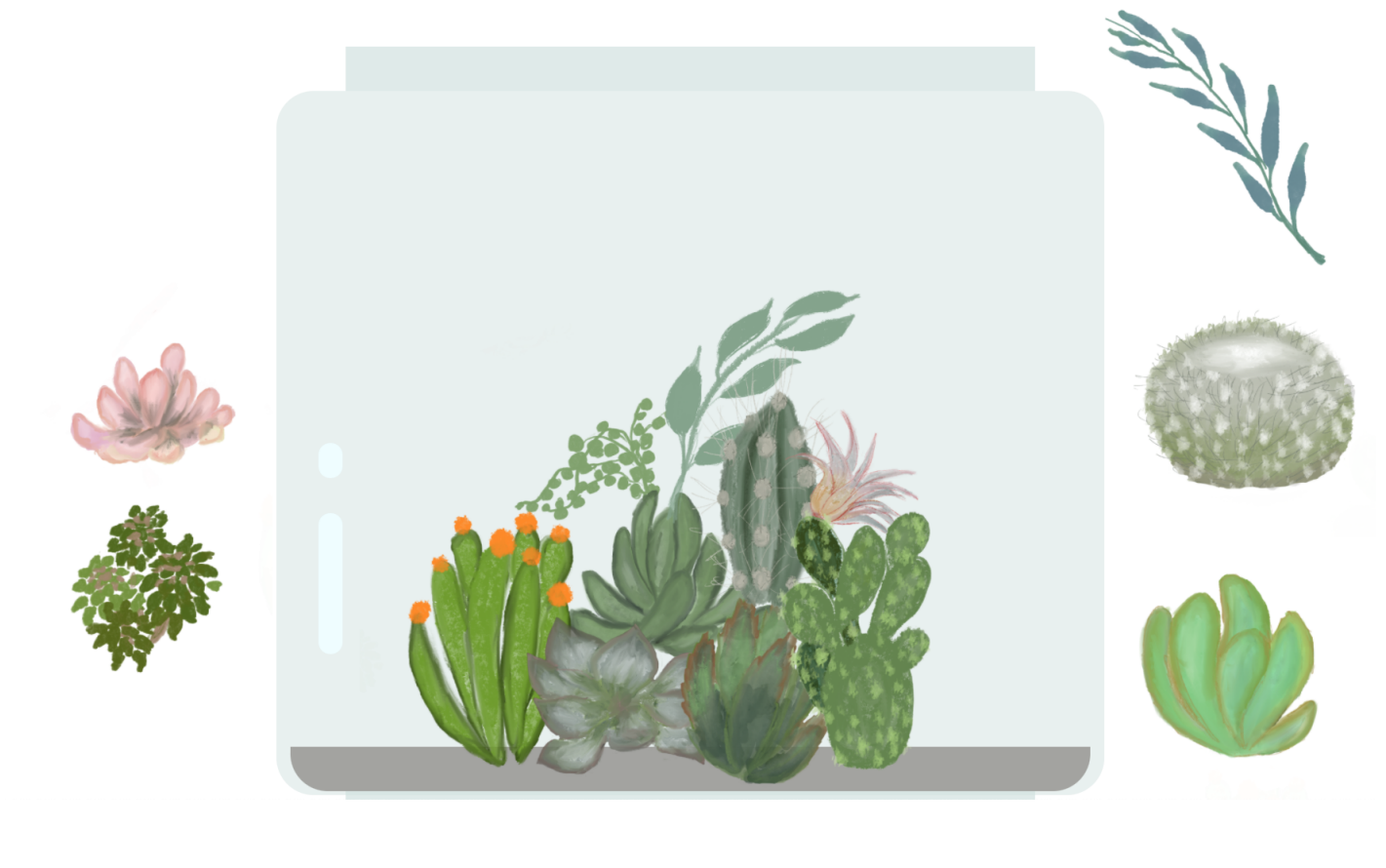
This small function resets the `onpointerup` and `onpointermove` events so that you can either restart your plant's progress by starting to drag it again, or start dragging a new plant.

✔ What happens if you don't set these events to null?

Now you have completed your project!

🎉 Congratulations! You have finished your beautiful terrarium.

My Terrarium



Challenge

Add new event handler to your closure to do something more to the plants; for example, double-click a plant to bring it to the front. Get creative!

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

While dragging elements around the screen seems trivial, there are many ways to do this and many pitfalls, depending on the effect you seek. In fact, there is an entire [drag and drop API](#) that you can try. We didn't use it in this module because the effect we wanted was somewhat different, but try this API on your own project and see what you can achieve.

Find more information on pointer events on the [W3C docs](#) and on [MDN web docs](#).

Always check browser capabilities using [CanIUse.com](#).

Assignment

[Work a bit more with the DOM](#)

Creating a game using events

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Event driven programming

When creating a browser based application, we provide a graphical user interface (GUI) for the user to use when interacting with what we've built. The most common way to interact with the browser is through clicking and typing in various elements. The challenge we face as a developer is we don't know when they're going to perform these operations!

Event-driven programming is the name for the type of programming we need to do to create our GUI. If we break this phrase down a little bit, we see the core word here is **event**. Event, according to Merriam-Webster, is defined as "something which happens". This describes our situation perfectly. We know something is going to happen for which we want to execute some code in response, but we don't know when it will take place.

The way we mark a section of code we want to execute is by creating a function. When we think about procedural programming, functions are called in a specific order. This same thing is going to be true with event driven programming. The difference is **how** the functions will be called.

To handle events (button clicking, typing, etc.), we register **event listeners**. An event listener is a function which listens for an event to occur and executes in response. Event listeners can update the UI, make calls to the server, or whatever else needs to be done in response to the user's action. We add an event listener by using addEventListener, and providing a function to execute.

NOTE: It's worth highlighting there are numerous ways to create event listeners. You can use anonymous functions, or create named ones. You can use various shortcuts, like setting the `click` property, or using `addEventListener`. In our exercise we are going to focus on `addEventListener` and anonymous functions, as it's probably the most common technique web developers use. It's also the most flexible, as `addEventListener` works for all events, and the event name can be provided as a parameter.

Common events

There are dozens of events available for you to listen to when creating an application. Basically anything a user does on a page raises an event, which gives you a lot of power to ensure they get the experience you desire. Fortunately, you'll normally only need a small handful of events. Here's a few common ones (including the two we'll use when creating our game):

- click: The user clicked on something, typically a button or hyperlink
- contextmenu: The user clicked the right mouse button
- select: The user highlighted some text
- input: The user input some text

Creating the game

We are going to create a game to explore how events work in JavaScript. Our game is going to test a player's typing skill, which is one of the most underrated skills all developers should have. We should all be practicing our typing! The general flow of the game will look like this:

- Player clicks on start button and is presented with a quote to type
- Player types the quote as quickly as they can in a textbox
 - As each word is completed, the next one is highlighted
 - If the player has a typo, the textbox is updated to red
 - When the player completes the quote, a success message is displayed with the elapsed time

Let's build our game, and learn about events!

File structure

We're going to need three total files: **index.html**, **script.js** and **style.css**. Let's start by setting those up to make life a little easier for us.

- Create a new folder for your work by opening a console or terminal window and issuing the following command:

bash

```
# Linux or macOS  
mkdir typing-game && cd typing-game
```

```
# Windows  
md typing-game && cd typing-game
```

- Open Visual Studio Code

bash

```
code .
```

- Add three files to the folder in Visual Studio Code with the following names:
 - index.html
 - script.js
 - style.css

Create the user interface

If we explore the requirements, we know we're going to need a handful of elements on our HTML page. This is sort of like a recipe, where we need some ingredients:

- Somewhere to display the quote for the user to type
- Somewhere to display any messages, like a success message
- A textbox for typing
- A start button

Each of those will need IDs so we can work with them in our JavaScript. We will also add references to the CSS and JavaScript files we're going to create.

Create a new file named **index.html**. Add the following HTML:

html

```
<!-- inside index.html -->
<html>
<head>
  <title>Typing game</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Typing game!</h1>
  <p>Practice your typing skills with a quote from Sherlock Holmes. Click >
  <p id="quote"></p> <!-- This will display our quote -->
  <p id="message"></p> <!-- This will display any status messages -->
  <div>
    <input type="text" aria-label="current word" id="typed-value" /> <!--
    <button type="button" id="start">Start</button> <!-- To start the game
  </div>
  <script src="script.js"></script>
</body>
</html>
```

Launch the application

It's always best to develop iteratively to see how things look. Let's launch our application. There's a wonderful extension for Visual Studio Code called [Live Server](#) which will both host your application locally and refresh the browser each time you save.

- Install [Live Server](#) by following the link and clicking **Install**
 - You will be prompted by the browser to open Visual Studio Code, and then by Visual Studio Code to perform the installation

- Restart Visual Studio Code if prompted
- Once installed, in Visual Studio Code, click Ctrl-Shift-P (or Cmd-Shift-P) to open the command palette
- Type **Live Server: Open with Live Server**
 - Live Server will start hosting your application
- Open a browser and navigate to <https://localhost:5500>
- You should now see the page you created!

Let's add some functionality.

Add the CSS

With our HTML created, let's add the CSS for core styling. We need to highlight the word the player should be typing, and colorize the textbox if what they've typed is incorrect. We'll do this with two classes.

Create a new file named **style.css** and add the following syntax.

```
/* inside style.css */
.highlight {
  background-color: yellow;
}

.error {
  background-color: lightcoral;
  border: red;
}
```

CSS

✅ When it comes to CSS you can layout your page however you might like. Take a little time and make the page look more appealing:

- Choose a different font
- Colorize the headers
- Resize items

JavaScript

With our UI created, it's time to focus our attention on the JavaScript which will provide the logic.

We're going to break this down into a handful of steps:

- [Create the constants](#)
- [Event listener to start the game](#)
- [Event listener to typing](#)

But first, create a new file named **script.js**.

Add the constants

We're going to need a few items to make our lives a little easier for programming. Again, similar to a recipe, here's what we'll need:

- Array with the list of all quotes
- Empty array to store all the words for the current quote
- Space to store the index of the word the player is currently typing
- The time the player clicked start

We're also going to want references to the UI elements:

- The textbox (**typed-value**)
- The quote display (**quote**)
- The message (**message**)

javascript

```
// inside script.js
// all of our quotes
const quotes = [
  'When you have eliminated the impossible, whatever remains, however im',
  'There is nothing more deceptive than an obvious fact.',
  'I ought to know by this time that when a fact appears to be opposed to',
  'I never make exceptions. An exception disproves the rule.',
  'What one man can invent another can discover.',
  'Nothing clears up a case so much as stating it to another person.',
  'Education never ends, Watson. It is a series of lessons, with the grea
];
// store the list of words and the index of the word the player is current
let words = [];
let wordIndex = 0;
// the starting time
let startTime = Date.now();
// page elements
const quoteElement = document.getElementById('quote');
```

```
const messageElement = document.getElementById('message');
const typedValueElement = document.getElementById('typed-value');
```

✔ Go ahead and add more quotes to your game

NOTE: We can retrieve the elements whenever we want in code by using `document.getElementById`. Because of the fact we're going to refer to these elements on a regular basis we're going to avoid typos with string literals by using constants. Frameworks such as Vue.js or React can help you better manage centralizing your code.

Take a minute to watch a video on using `const`, `let` and `var`



🎥 Click the image above for a video about variables.

Add start logic

To begin the game, the player will click on start. Of course, we don't know when they're going to click start. This is where an event listener comes into play. An event listener will allow us to listen for something to occur (an event) and execute code in response. In our case, we want to execute code when the user clicks on start.

When the user clicks **start**, we need to select a quote, setup the user interface, and setup tracking for the current word and timing. Below is the JavaScript you'll need to add; we discuss it just after the script block.

javascript

```
// at the end of script.js
document.getElementById('start').addEventListener('click', () => {
  // get a quote
  const quoteIndex = Math.floor(Math.random() * quotes.length);
  const quote = quotes[quoteIndex];
  // Put the quote into an array of words
  words = quote.split(' ');
  // reset the word index for tracking
  wordIndex = 0;

  // UI updates
  // Create an array of span elements so we can set a class
  const spanWords = words.map(function(word) { return `${word} `});
  // Convert into string and set as innerHTML on quote display
  quoteElement.innerHTML = spanWords.join('');
  // Highlight the first word
  quoteElement.childNodes[0].className = 'highlight';
  // Clear any prior messages
  messageElement.innerText = '';

  // Setup the textbox
  // Clear the textbox
  typedValueElement.value = '';
  // set focus
  typedValueElement.focus();
  // set the event handler

  // Start the timer
  startTime = new Date().getTime();
});
```

Let's break down the code!

- Setup the word tracking
 - Using `Math.floor` and `Math.random` allows us to randomly select a quote from the `quotes` array
 - We convert the `quote` into an array of `words` so we can track the word the player is currently typing
 - `wordIndex` is set to 0, since the player will start on the first word

- Setup the UI
 - Create an array of `spanWords`, which contains each word inside a `span` element
 - This will allow us to highlight the word on the display
 - `join` the array to create a string which we can use to update the `innerHTML` on `quoteElement`
 - This will display the quote to the player
 - Set the `className` of the first `span` element to `highlight` to highlight it as yellow
 - Clean the `messageElement` by setting `innerText` to ''
- Setup the textbox
 - Clear the current `value` on `typedValueElement`
 - Set the `focus` to `typedValueElement`
- Start the timer by calling `getTime`

Add typing logic

As the player types, an `input` event will be raised. This event listener will check to ensure the player is typing the word correctly, and handle the current status of the game. Returning to **script.js**, add the following code to the end. We will break it down afterwards.

javascript

```
// at the end of script.js
typedValueElement.addEventListener('input', () => {
  // Get the current word
  const currentWord = words[wordIndex];
  // get the current value
  const typedValue = typedValueElement.value;

  if (typedValue === currentWord && wordIndex === words.length - 1) {
    // end of sentence
    // Display success
    const elapsedTime = new Date().getTime() - startTime;
    const message = `CONGRATULATIONS! You finished in ${elapsedTime / 1000}`;
    messageElement.innerText = message;
  } else if (typedValue.endsWith(' ') && typedValue.trim() === currentWord) {
    // end of word
    // clear the typedValueElement for the new word
    typedValueElement.value = '';
    // move to the next word
    wordIndex++;
    // reset the class name for all elements in quote
    for (const wordElement of quoteElement.childNodes) {
      wordElement.className = '';
    }
  }
});
```

```

    }
    // highlight the new word
    quoteElement.childNodes[wordIndex].className = 'highlight';
  } else if (currentWord.startsWith(typedValue)) {
    // currently correct
    // highlight the next word
    typedValueElement.className = '';
  } else {
    // error state
    typedValueElement.className = 'error';
  }
});

```

Let's break down the code! We start by grabbing the current word and the value the player has typed thus far. Then we have waterfall logic, where we check if the quote is complete, the word is complete, the word is correct, or (finally), if there is an error.

- Quote is complete, indicated by `typedValue` being equal to `currentWord`, and `wordIndex` being equal to one less than the `length` of `words`
 - Calculate `elapsedTime` by subtracting `startTime` from the current time
 - Divide `elapsedTime` by 1,000 to convert from milliseconds to seconds
 - Display a success message
- Word is complete, indicated by `typedValue` ending with a space (the end of a word) and `typedValue` being equal to `currentWord`
 - Set `value` on `typedElement` to be `' '` to allow for the next word to be typed
 - Increment `wordIndex` to move to the next word
 - Loop through all `childNodes` of `quoteElement` to set `className` to `' '` to revert to default display
 - Set `className` of the current word to `highlight` to flag it as the next word to type
- Word is currently typed correctly (but not complete), indicated by `currentWord` started with `typedValue`
 - Ensure `typedValueElement` is displayed as default by clearing `className`
- If we made it this far, we have an error
 - Set `className` on `typedValueElement` to `error`

Test your application

You've made it to the end! The last step is to ensure our application works. Give it a shot! Don't worry if there are errors; **all developers** have errors. Examine the messages and debug as needed.

Click on **start**, and start typing away! It should look a little like the animation we saw before.

Practice your typing

Click start to have a quote displayed. Type the quote as fast as you can!

Challenge

Add more functionality

- Disable the `input` event listener on completion, and re-enable it when the button is clicked
- Disable the textbox when the player completes the quote
- Display a modal dialog box with the success message
- Store high scores using `localStorage`

Post-Lecture Quiz

[Post-lecture quiz](#)

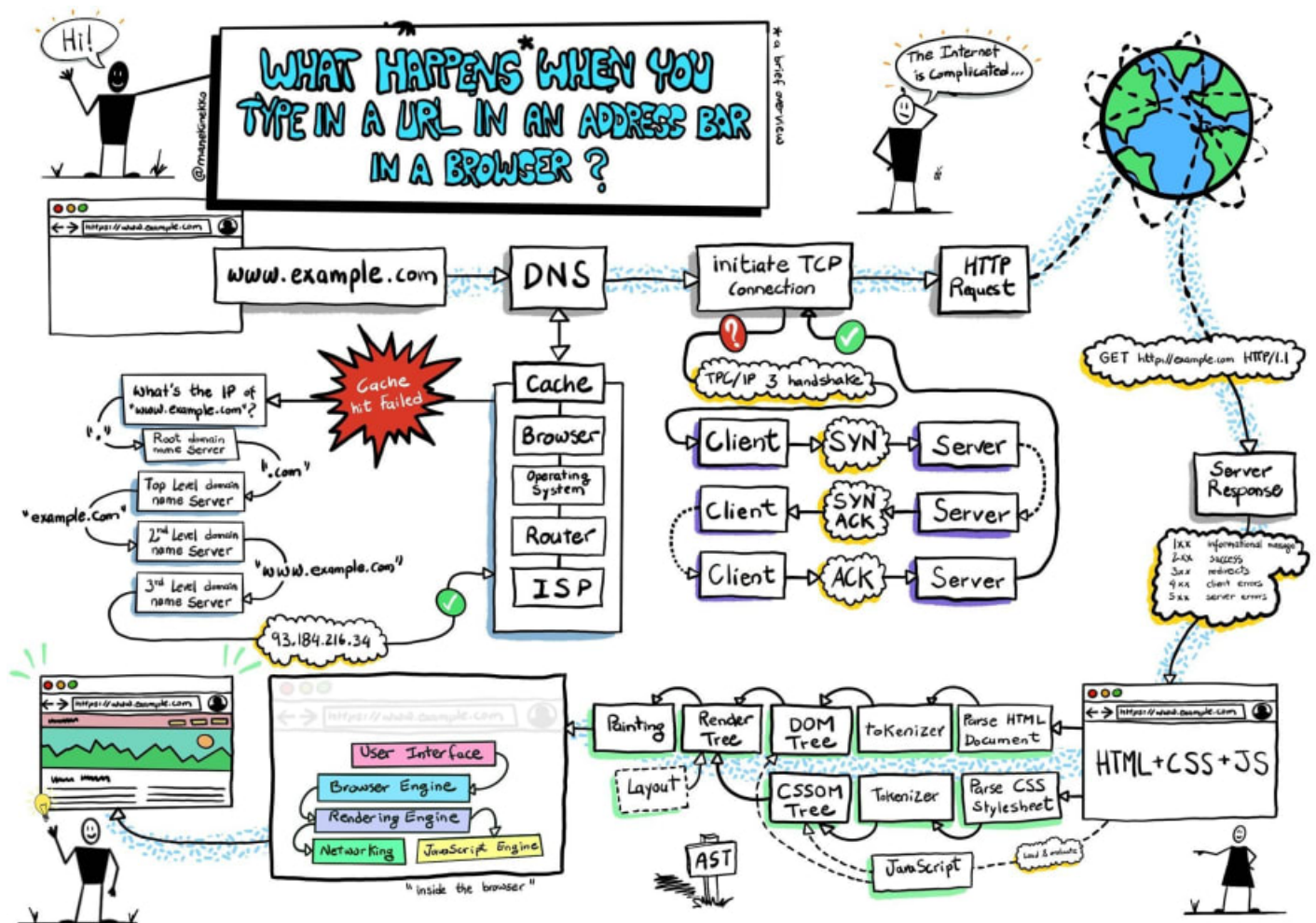
Review & Self Study

Read up on [all the events available](#) to the developer via the web browser, and consider the scenarios in which you would use each one.

Assignment

Create a new keyboard game

Browser Extension Project Part 1: All about Browsers



Sketchnote by [Wassim Chegham](#)

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

Browser extensions add additional functionality to a browser. But before you build one, you should learn a little about how browsers do their work.

About the browser

In this series of lessons, you'll learn how to build a browser extension that will work on Chrome, Firefox and Edge browsers. In this part, you'll discover how browsers work and scaffold out the elements of the browser extension.

But what is a browser exactly? It is a software application that allows an end user to access content from a server and display it on web pages.

✅ A little history: the first browser was called 'WorldWideWeb' and was created by Sir Timothy Berners-Lee in 1990.

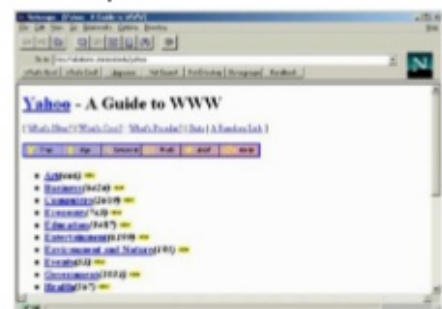
WORLD WIDE WEB (1990s)



First Web Browser/Editor, 1990



Netscape, 1994



Yahoo, 1994

Some early browsers, via [Karen McGrane](#)

When a user connected to the internet using a URL (Uniform Resource Locator) address, usually using Hypertext Transfer Protocol via an `http` or `https` address, the browser communicates with a web server and fetches a web page.

At this point, the browser's rendering engine displays it on the user's device, which might be a mobile phone, desktop, or laptop.

Browsers also have the ability to cache content so that it doesn't have to be retrieved from the server every time. They can record the history of a user's browsing activity, store 'cookies', which are small bits of data that contain information used to store a user's activity, and more.

A really important thing to remember about browsers is that they are not all the same! Each browser has its strengths and weaknesses, and a professional web developer needs to understand how to make web pages perform well cross-browser. This includes handling small viewports such as a mobile phone's, as well as a user who is offline.

A really useful website that you probably should bookmark in whatever browser you prefer to use is caniuse.com. When you are building web pages, it's very helpful to use caniuse's lists of supported technologies so that you can best support your users.

✅ How can you tell what browsers are most popular with your web site's user base? Check your analytics - you can install various analytics packages as part of your web development process, and they will tell you what browsers are most used by the various popular browsers.

Browser extensions

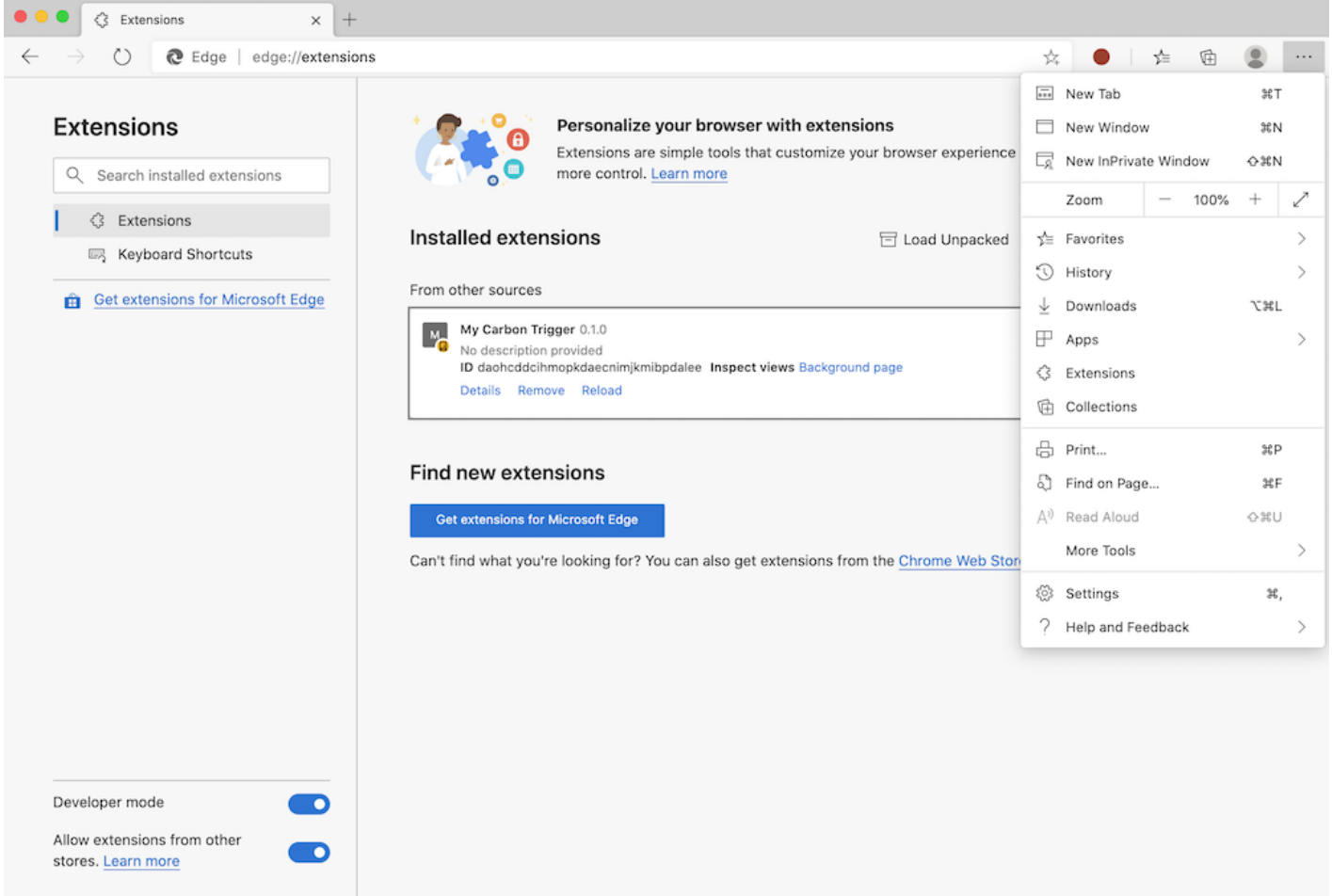
Why would you want to build a browser extension? It's a handy thing to attach to your browser when you need quick access to tasks that you tend to repeat. For example, if you find yourself needing to check colors on the various web pages that you interact with, you might install a color-picker browser extension. If you have trouble remembering passwords, you might use a password-management browser extension.

Browser extensions are fun to develop, too. They tend to manage a finite number of tasks that they perform well.

✅ What are your favorite browser extensions? What tasks do they perform?

Installing extensions

Before you start building, take a look at the process of building and deploying a browser extension. While each browser varies a bit in how they manage this task, the process is similar on Chrome and Firefox to this example on Edge:



In essence, the process will be:

- build your extension using `npm run build`
- navigate in the browser to the extensions pane using the "Settings and more" button (the `⋮` icon) on the top right
- if it's a new installation, choose `load unpacked` to upload a fresh extension from its build folder (in our case it is `/dist`)
- or, click `reload` if you are reloading the already-installed extension

✅ These instructions pertain to extensions you build yourself; to install extensions that have been released to the browser extension store associated to each browser, you should navigate to those [stores](#) and install the extension of your choice.

Get Started

You're going to build a browser extension that displays your region's carbon footprint, showing your region's energy usage and the source of the energy. The extension will have a form that collects an API key so that you can access CO2 Signal's API.

You need:

- [an API key](#); enter your email in the box on this page and one will be sent to you
- the [code for your region](#) corresponding to the [Electricity Map](#) (in Boston, for example, I use 'US-NEISO').

- the [starter code](#). Download the `start` folder; you will be completing code in this folder.
- [NPM](#) - NPM is a package management tool; install it locally and the packages listed in your `package.json` file will be installed for use by your web asset

✔ Learn more about package management in this [excellent Learn module](#)

Take a minute to look through the codebase:

`dist` -|`manifest.json` (defaults set here) -|`index.html` (front-end HTML markup here) -|`background.js` (background JS here) -|`main.js` (built JS) `src` -|`index.js` (your JS code goes here)

✔ Once you have your API key and Region code handy, store those somewhere in a note for future use.

Build the HTML for the extension

This extension has two views. One to gather the API key and region code:

Personalize your browser
Extensions are simple tools that give you more control. [Learn more](#)

extensions


sources

Carbon Trigger 0.1.0
Description provided
gpdgghjmdiajokadkdlfnjgmahcd Inspect view
s Remove Reload

extensions

Extensions for Microsoft Edge

What are you looking for? You can also get



Welcome to Your Personal Carbon Trigger!

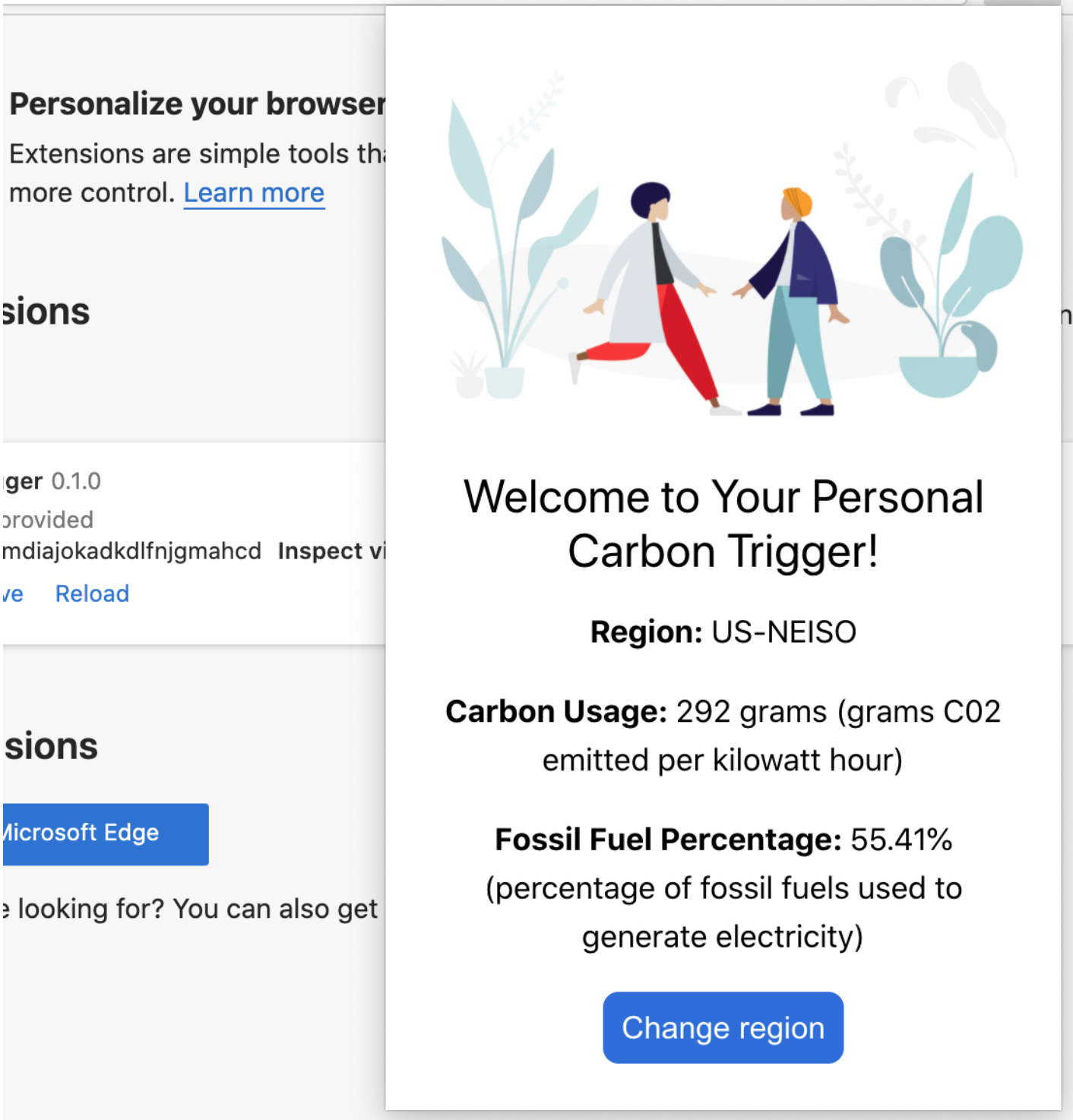
New? Add your Information

Region Name

Your API Key from tmrow

Submit

And the second to display the region's carbon usage:



Let's start by building the HTML for the form and styling it with CSS.

In the `/dist` folder, you will build a form and a result area. In the `index.html` file, populate the delineated form area:

HTML

```
<form class="form-data" autocomplete="on">
  <div>
    <h2>New? Add your Information</h2>
  </div>
  <div>
    <label for="region">Region Name</label>
    <input type="text" id="region" required class="region-name" />
  </div>
</form>
```



```

</div>
<div>
  <label for="api">Your API Key from tmrow</label>
  <input type="text" id="api" required class="api-key" />
</div>
<button class="search-btn">Submit</button>
</form>

```

This is the form where your saved information will be input and saved to local storage.

Next, create the results area; under the final form tag, add some divs:

HTML

```

<div class="result">
  <div class="loading">loading...</div>
  <div class="errors"></div>
  <div class="data"></div>
  <div class="result-container">
    <p><strong>Region: </strong><span class="my-region"></span></p>
    <p><strong>Carbon Usage: </strong><span class="carbon-usage"></span></p>
    <p><strong>Fossil Fuel Percentage: </strong><span class="fossil-fuel"></span></p>
  </div>
  <button class="clear-btn">Change region</button>
</div>

```

At this point, you can try a build. Make sure to install the package dependencies of this extension:

```
npm install
```

This command will use npm, the Node Package Manager, to install webpack for your extension's build process. Webpack is a bundler that handles compiling code. You can see the output of this process by looking in `/dist/main.js` - you see the code has been bundled.

For now, the extension should build and, if you deploy it into Edge as an extension, you'll see a form neatly displayed.

Congratulations, you've taken the first steps towards building a browser extension. In subsequent lessons, you'll make it more functional and useful.

Take a look at a browser extension store and install one to your browser. You can examine its files in interesting ways. What do you discover?

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

In this lesson you learned a little about the history of the web browser; take this opportunity to learn about how the inventors of the World Wide Web envisioned its use by reading more about its history. Some useful sites include:

[The History of Web Browsers](#)

[History of the Web](#)

[An interview with Tim Berners-Lee](#)

Assignment

[Restyle your extension](#)

Browser Extension Project Part 2: Call an API, use Local Storage

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

In this lesson, you'll call an API by submitting your browser extension's form and display the results in your browser extension. In addition, you'll learn about how you can store data in your browser's local storage for future reference and use.

✔ Follow the numbered segments in the appropriate files to know where to place your code

Set up the elements to manipulate in the extension:

By this time you have built the HTML for the form and results `<div>` for your browser extension. From now on, you'll need to work in the `/src/index.js` file and building your extension bit by bit. Refer to the [previous lesson](#) on getting your project set up and on the build process.

Working in your `index.js` file, start by creating some `const` variables to hold the values associated with various fields:

JavaScript

```
// form fields
const form = document.querySelector('.form-data');
const region = document.querySelector('.region-name');
const apiKey = document.querySelector('.api-key');

// results
const errors = document.querySelector('.errors');
const loading = document.querySelector('.loading');
const results = document.querySelector('.result-container');
const usage = document.querySelector('.carbon-usage');
const fossilfuel = document.querySelector('.fossil-fuel');
const myregion = document.querySelector('.my-region');
const clearBtn = document.querySelector('.clear-btn');
```

All of these fields are referenced by their css class, as you set it up in the HTML in the previous lesson.

Add listeners

Next, add event listeners to the form and the clear button that resets the form, so that if a user submits the form or clicks that reset button, something will happen, and add the call to initialize the app at the bottom of the file:

JavaScript

```
form.addEventListener('submit', (e) => handleSubmit(e));
clearBtn.addEventListener('click', (e) => reset(e));
init();
```

✔ Notice the shorthand used to listen for a submit or click event, and how the event it is passed to the `handleSubmit` or `reset` functions. Can you write the equivalent of this shorthand in a longer format? Which do you prefer?

Build out the `init()` function and the `reset()` function:

Now you are going to build the function that initializes the extension, which is called `init()`:

JavaScript

```
function init() {
  //if anything is in localStorage, pick it up
  const storedApiKey = localStorage.getItem('apiKey');
  const storedRegion = localStorage.getItem('regionName');

  //set icon to be generic green
  //todo

  if (storedApiKey === null || storedRegion === null) {
    //if we don't have the keys, show the form
    form.style.display = 'block';
    results.style.display = 'none';
    loading.style.display = 'none';
    clearBtn.style.display = 'none';
    errors.textContent = '';
  } else {
    //if we have saved keys/regions in localStorage, show results when
    displayCarbonUsage(storedApiKey, storedRegion);
    results.style.display = 'none';
    form.style.display = 'none';
    clearBtn.style.display = 'block';
  }
};

function reset(e) {
  e.preventDefault();
  //clear local storage for region only
  localStorage.removeItem('regionName');
  init();
}
```

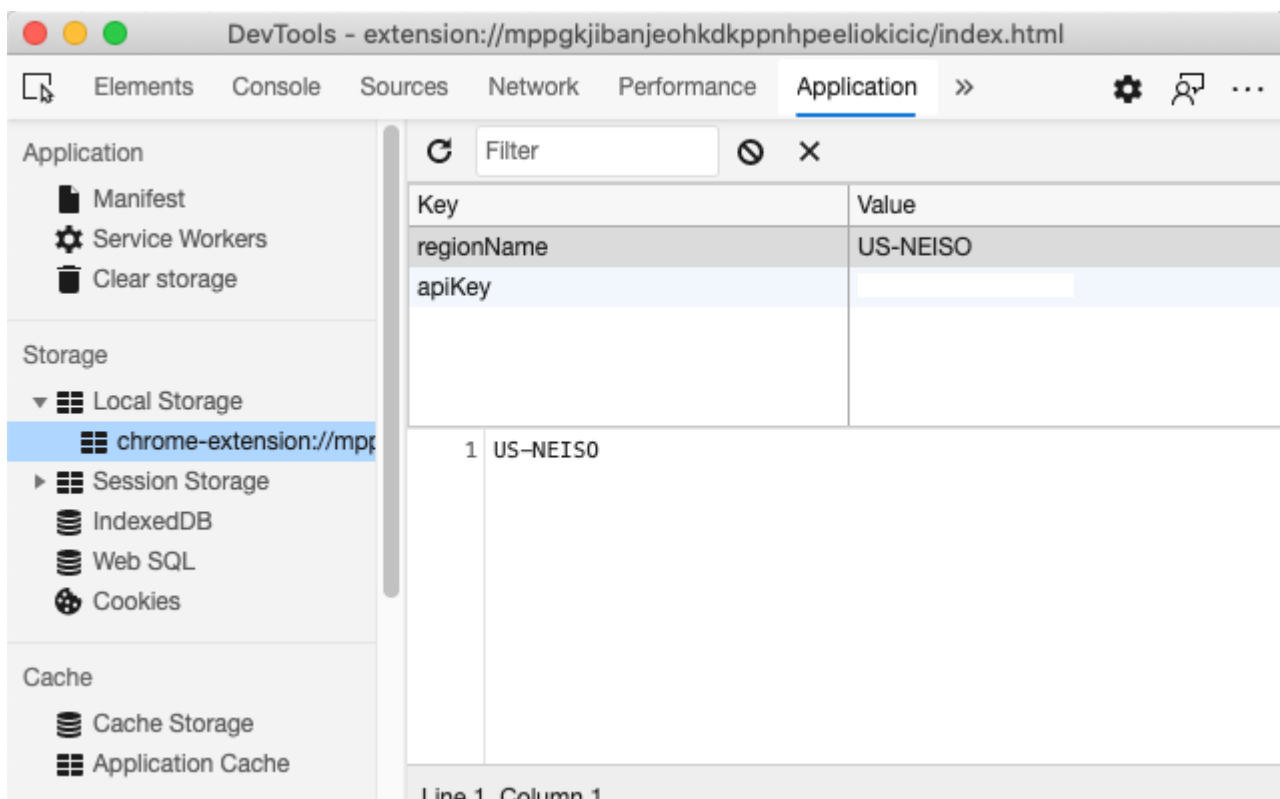
In this function, there is some interesting logic. Reading through it, can you see what happens?

- two `const` are set up to check if the user has stored an APIKey and region code in local storage.
- if either of those is null, show the form by changing its style to display as 'block'
- hide the results, loading, and clearBtn and set any error text to an empty string
- if there exists a key and region, start a routine to:
 - call the API to get carbon usage data
 - hide the results area
 - hide the form
 - show the reset button

Before moving on, it's useful to learn about a very important concept available in browsers: LocalStorage. LocalStorage is a useful way to store strings in the browser as a `key-value` pair. This type of web storage can be manipulated by JavaScript to manage data in the browser. LocalStorage does not expire, while SessionStorage, another kind of web storage, is cleared when the browser is closed. The various types of storage have pros and cons to their usage.

Note - your browser extension has its own local storage; the main browser window is a different instance and behaves separately.

You set your APIKey to have a string value, for example, and you can see that it is set on Edge by "inspecting" a web page (you can right-click a browser to inspect) and going to the Applications tab to see the storage.



✔ Think about situations where you would NOT want to store some data in LocalStorage. In general, placing API Keys in LocalStorage is a bad idea! Can you see why? In our case, since our app is purely for learning and will not be deployed to an app store, we will use this method.

Notice that you use the Web API to manipulate LocalStorage, either by using `getItem()`, `setItem()` or `removeItem()`. It's widely supported across browsers.

Before building the `displayCarbonUsage()` function that is called in `init()`, let's build the functionality to handle the initial form submission.

Handle the form submission

Create a function called `handleSubmit` that accepts an event argument (`e`). Stop the event from propagating (in this case, we want to stop the browser from refreshing) and call a new function, `setUpUser`, passing in the arguments `apiKey.value` and `region.value`. In this way, you use the two values that are brought in via the initial form when the appropriate fields are populated.

JavaScript

```
function handleSubmit(e) {
  e.preventDefault();
  setUpUser(apiKey.value, region.value);
}
```

✔ Refresh your memory - the HTML you set up in the last lesson has two input fields whose `values` are captured via the `const` you set up at the top of the file, and they are both `required` so the browser stops users from inputting null values.

Set up the user

Moving on to the `setUpUser` function, here is where you set local storage values for `apiKey` and `regionName`. Add a new function:

JavaScript

```
function setUpUser(apiKey, regionName) {
  localStorage.setItem('apiKey', apiKey);
  localStorage.setItem('regionName', regionName);
  loading.style.display = 'block';
  errors.textContent = '';
  clearBtn.style.display = 'block';
  //make initial call
  displayCarbonUsage(apiKey, regionName);
}
```

This function sets a loading message to show while the API is called. At this point, you have arrived at creating the most important function of this browser extension!

Display Carbon Usage

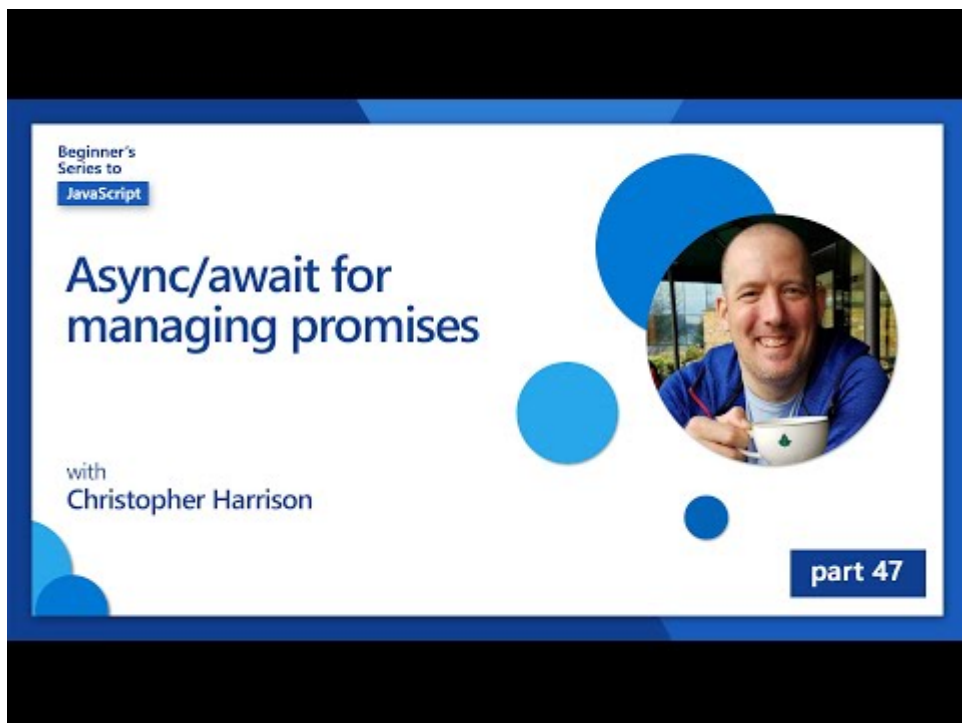
Finally it's time to query the API!

Before going further, we should discuss APIs. APIs, or Application Programming Interfaces, are a critical element of a web developer's toolbox. They provide standard ways for programs to interact and interface with each other. For example, if you are building a web site that needs to query a database, someone might have created an API for you to use. While there are many types of APIs, one of the most popular is a REST API.

✅ The term 'REST' stands for 'Representational State Transfer' and features using variously-configured URLs to fetch data. Do a little research on the various types of APIs available to developers. What format appeals to you?

There are important things to note about this function. First notice the async keyword. Writing your functions so that they run asynchronously means that they wait for an action, such as data being returned, to be completed before continuing.

Here's a quick video about `async` :



📺 Click the image above for a video about `async/await`.

Create a new function to query the C02Signal API:

```

import axios from '../node_modules/axios';

async function displayCarbonUsage(apiKey, region) {
  try {
    await axios
      .get('https://api.co2signal.com/v1/latest', {
        params: {
          countryCode: region,
        },
        headers: {
          'auth-token': apiKey,
        },
      })
      .then((response) => {
        let CO2 = Math.floor(response.data.data.carbonIntensity);

        //calculateColor(CO2);

        loading.style.display = 'none';
        form.style.display = 'none';
        myregion.textContent = region;
        usage.textContent =
          Math.round(response.data.data.carbonIntensity) + ' gram
        fossilfuel.textContent =
          response.data.data.fossilFuelPercentage.toFixed(2) +
          '% (percentage of fossil fuels used to generate electr:
        results.style.display = 'block';
      });
  } catch (error) {
    console.log(error);
    loading.style.display = 'none';
    results.style.display = 'none';
    errors.textContent = 'Sorry, we have no data for the region you ha
  }
}

```

This is a big function. What's going on here?

- following best practices, you use an `async` keyword to make this function behave asynchronously. The function contains a `try/catch` block as it will return a promise when the API returns data. Because you don't have control over the speed that the API will respond (it may not respond at all!), you need to handle this uncertainty by call it asynchronously.

- you're querying the co2signal API to get your region's data, using your API Key. To use that key, you have to use a type of authentication in your header parameters.
- once the API responds, you assign various elements of its response data to the parts of your screen you set up to show this data.
- if there's an error, or if there is no result, you show an error message.

✔ Using asynchronous programming patterns is another very useful tool in your toolbox. Read [about the various ways](#) you can configure this type of code.

Congratulations! If you build your extension (`npm run build`) and refresh it in your extensions pane, you have a working extension! The only thing that isn't working is the icon, and you'll fix that in the next lesson.

Challenge

We've discussed several types of API so far in these lessons. Choose a web API and research in depth what it offers. For example, take a look at APIs available within browsers such as the [HTML Drag and Drop API](#). What makes a great API in your opinion?

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

You learned about LocalStorage and APIs in this lesson, both very useful for the professional web developer. Can you think how these two things work together? Think about how you would architect a web site that would store items to be used by an API.

Assignment

[Adopt an API](#)

Browser Extension Project Part 3: Learn about Background Tasks and Performance

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

In the last two lessons of this module, you learned how to build a form and display area for data fetched from an API. It's a very standard way of creating web presences on the web. You even learned how to handle fetching data asynchronously. Your browser extension is very nearly complete.

It remains to manage some background tasks, including refreshing the color of the extension's icon, so this is a great time to talk about how the browser manages this kind of task. Let's think about these browser tasks in the context of the performance of your web assets as you build them.

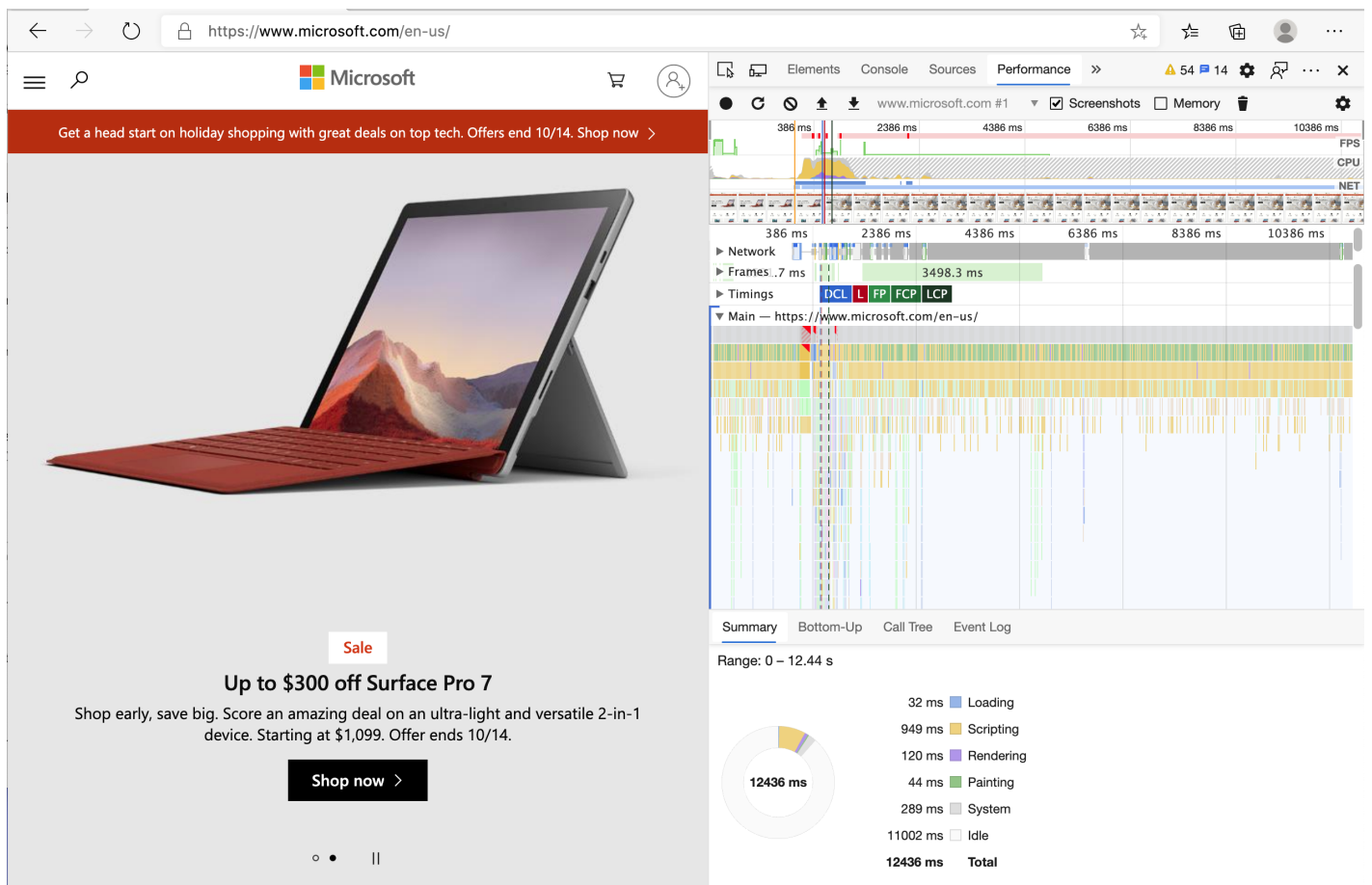
Web Performance Basics

"Website performance is about two things: how fast the page loads, and how fast the code on it runs." -- [Zack Grossbart](#)

The topic of how to make your web sites blazingly fast on all kinds of devices, for all kinds of users, in all kinds of situations, is unsurprisingly vast. Here are some points to keep in mind as you build either a standard web project or a browser extension.

The first thing you need to do to ensure that your site is running efficiently is to gather data about its performance. The first place to do this is in the developer tools of your web browser. In Edge, you can select the "Settings and more" button (the three dots icon on the top right of the browser), then navigate to More Tools > Developer Tools and open the Performance tab. You can also use the keyboard shortcuts `Ctrl + Shift + I` on Windows, or `Option + Command + I` on Mac to open developer tools.

The Performance tab contains a Profiling tool. Open a web site (try, for example, <https://www.microsoft.com>) and click the 'Record' button, then refresh the site. Stop the recording at any time, and you will be able to see the routines that are generated to 'script', 'render', and 'paint' the site:



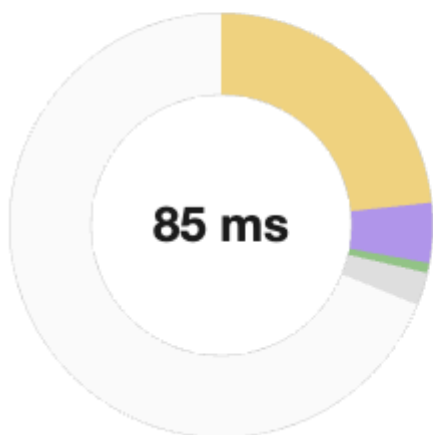
✔ Visit the [Microsoft Documentation](#) on the Performance panel in Edge

Tip: to get a true reading of your web site's startup time, clear your browser's cache

Select elements of the profile timeline to zoom in on events that happen while your page loads.

Get a snapshot of your page's performance by selecting a part of the profile timeline and looking at the summary pane:

Range: 0 – 85 ms



- 20 ms Scripting
- 4 ms Rendering
- 1 ms Painting
- 2 ms System
- 58 ms Idle
- 85 ms Total**

Check the Event Log pane to see if any event took longer than 15 ms:

Start Time	Self Time	Total Time	Activity
30.2 ms	0.1 ms	19.7 ms	Timer Fired

Summary Bottom-Up Call Tree **Event Log**

Filter Loading Scripting Rendering Painting

Timer Fired
Timer ID 62474

✔ Get to know your profiler! Open the developer tools on this site and see if there are any bottlenecks. What's the slowest-loading asset? The fastest?

Profiling checks

In general there are some "problem areas" that every web developer should watch for when building a site, so as to avoid nasty surprises when it's time to deploy to production.

Asset sizes: The web has gotten 'heavier', and thus slower, over the past few years. Some of this weight has to do with the use of images.

✔ Look through the [Internet Archive](#) for a historical view of page weight and more.

A good practice is to ensure that your images are optimized, delivered at the right size and resolution for your users.

DOM traversals: The browser has to build its Document Object Model based on the code you write, so it's in the interest of good page performance to keep your tags minimal, only using and styling

what the page needs. To this point, excess CSS associated with a page could be optimized; styles that need to be used only on one page don't need to be included in the main style sheet, for example.

JavaScript: Every JavaScript developer should watch for 'render-blocking' scripts that must be loaded before the rest of the DOM can be traversed and painted to the browser. Consider using `defer` with your inline scripts (as is done in the Terrarium module).

✔ Try some sites on a [Site Speed Test website](#) to learn more about the common checks that are done to determine site performance.

Now that you have an idea on how the browser renders the assets you send to it, let's look at the last few things you need to do to complete your extension:

Create a function to calculate color

Working in `/src/index.js`, add a function called `calculateColor()` after the series of `const` variables you set to gain access to the DOM:

JavaScript

```
function calculateColor(value) {
  let co2Scale = [0, 150, 600, 750, 800];
  let colors = ['#2AA364', '#F5EB4D', '#9E4229', '#381D02', '#381D02'];

  let closestNum = co2Scale.sort((a, b) => {
    return Math.abs(a - value) - Math.abs(b - value);
  })[0];
  console.log(value + ' is closest to ' + closestNum);
  let num = (element) => element > closestNum;
  let scaleIndex = co2Scale.findIndex(num);

  let closestColor = colors[scaleIndex];
  console.log(scaleIndex, closestColor);

  chrome.runtime.sendMessage({ action: 'updateIcon', value: { color: clo:
}
```

What's going on here? You pass in a value (the carbon intensity) from the API call you completed in the last lesson, and then you calculate how close its value is to the index presented in colors array. Then you send that closest color value over to the chrome runtime.

The `chrome.runtime` has [an API](#) that handles all kinds of background tasks, and your extension is leveraging that:

"Use the `chrome.runtime` API to retrieve the background page, return details about the manifest, and listen for and respond to events in the app or extension lifecycle. You can also use this API to convert the relative path of URLs to fully-qualified URLs."

✅ If you're developing this browser extension for Edge, it might surprise you that you're using a chrome API. The newer Edge browser versions run on the Chromium browser engine, so you can leverage these tools.

Note, if you want to profile a browser extension, launch the dev tools from within the extension itself, as it is its own separate browser instance.

Set a default icon color

Now, in the `init()` function, set the icon to be generic green to start by again calling chrome's `updateIcon` action:

JavaScript

```
chrome.runtime.sendMessage({
  action: 'updateIcon',
  value: {
    color: 'green',
  },
});
```

Call the function, execute the call

Next, call that function you just created by adding it to the promise returned by the `C02Signal` API:

JavaScript

```
//let C02...
calculateColor(C02);
```

And finally, in `/dist/background.js`, add the listener for these background action calls:

JavaScript

```
chrome.runtime.onMessage.addListener(function (msg, sender, sendResponse) {
  if (msg.action === 'updateIcon') {
    chrome.browserAction.setIcon({ imageData: drawIcon(msg.value) });
  }
});
```

```
    }  
  });  
  //borrowed from energy lollipop extension, nice feature!  
  function drawIcon(value) {  
    let canvas = document.createElement('canvas');  
    let context = canvas.getContext('2d');  
  
    context.beginPath();  
    context.fillStyle = value.color;  
    context.arc(100, 100, 50, 0, 2 * Math.PI);  
    context.fill();  
  
    return context.getImageData(50, 50, 100, 100);  
  }  
}
```

In this code, you are adding a listener for any messages coming to the backend task manager. If it's called 'updateIcon', then the next code is run, to draw an icon of the proper color using the Canvas API.

✅ You'll learn more about the Canvas API in the [Space Game lessons](#).

Now, rebuild your extension (`npm run build`), refresh and launch your extension, and watch the color change. Is it a good time to run an errand or wash the dishes? Now you know!

Congratulations, you've built a useful browser extension and learned more about how the browser works and how to profile its performance.

Challenge

Investigate some open source web sites have been around a long time ago, and, based on their GitHub history, see if you can determine how they were optimized over the years for performance, if at all. What is the most common pain point?

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

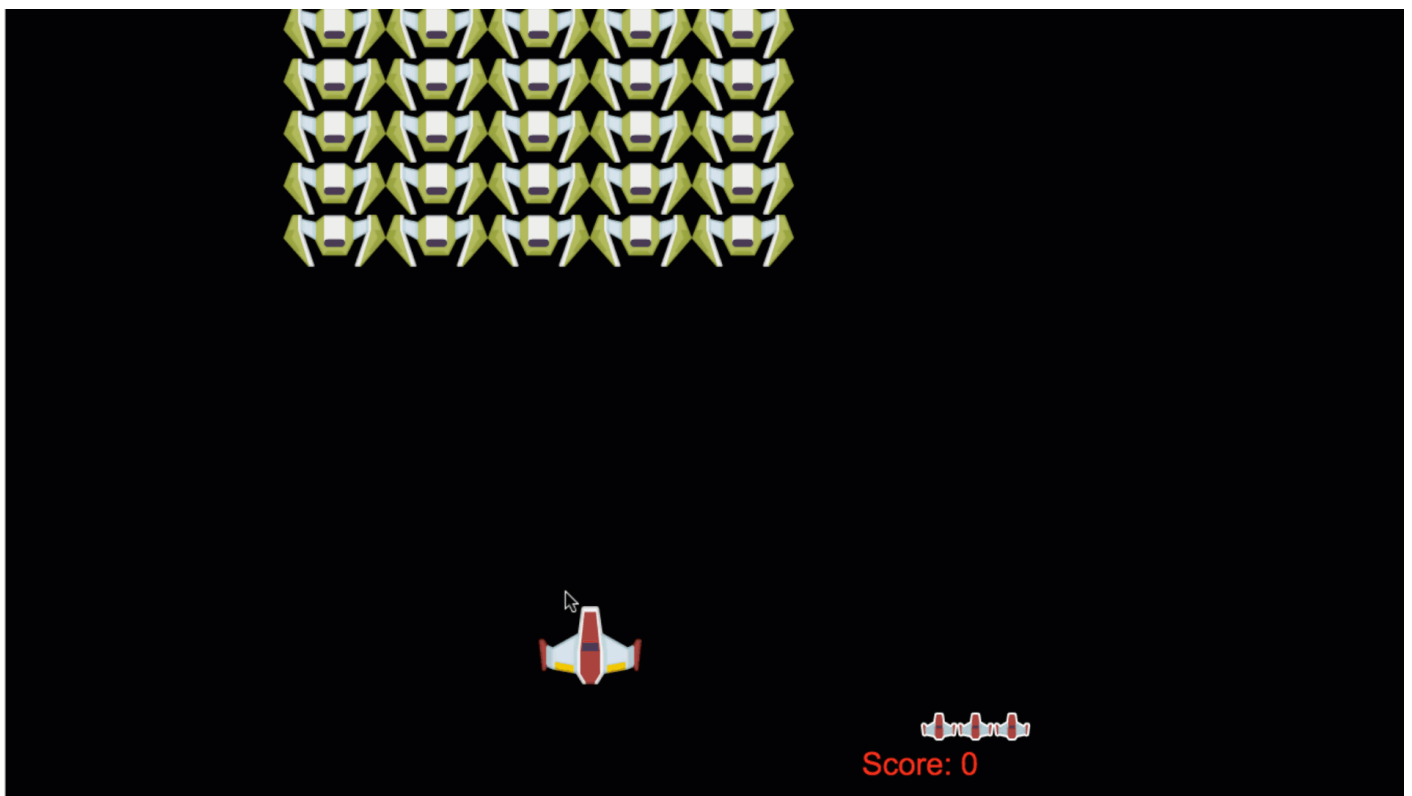
Consider signing up for a [performance newsletter](#)

Investigate some of the ways that browsers gauge web performance by looking through the performance tabs in their web tools. Do you find any major differences?

Assignment

[Analyze a site for performance](#)

Build a Space Game Part 1: Introduction



Pre-Lecture Quiz

[Pre-lecture quiz](#)

Inheritance and Composition in game development

In earlier lessons, there was not much need to worry about the design architecture of the apps you built, as the projects were very small in scope. However, when your applications grow in size and scope, architectural decisions become a larger concern. There are two major approaches to creating larger applications in JavaScript: *composition* or *inheritance*. There are pros and cons to both but let's explain them from within the context of a game.

✅ One of the most famous programming books ever written has to do with [design patterns](#).

In a game you have `game objects` which are objects that exist on a screen. This means they have a location on a cartesian coordinate system, characterized by having an `x` and `y` coordinate. As you develop a game you will notice that all your game objects have a standard property, common for every game you create, namely elements that are:

- **location-based** Most, if not all, game elements are location based. This means that they have a location, an `x` and `y`.
- **movable** These are objects that can move to a new location. This is typically a hero, a monster or an NPC (a non player character), but not for example, a static object like a tree.
- **self-destructing** These objects only exist for a set period of time before they set themselves up for deletion. Usually this is represented by a `dead` or `destroyed` boolean that signals to the game engine that this object should no longer be rendered.
- **cool-down** 'Cool-down' is a typical property among short-lived objects. A typical example is a piece of text or graphical effect like an explosion that should only be seen for a few milliseconds.

✅ Think about a game like Pac-Man. Can you identify the four object types listed above in this game?

Expressing behavior

All we described above are behavior that game objects can have. So how do we encode those? We can express this behavior as methods associated to either classes or objects.

Classes

The idea is to use `classes` in conjunction with `inheritance` to accomplish adding a certain behavior to a class.

✅ Inheritance is an important concept to understand. Learn more on [MDN's article about inheritance](#).

Expressed via code, a game object can typically look like this:

javascript

```
//set up the class GameObject
class GameObject {
```

```

    constructor(x, y, type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }
}

//this class will extend the GameObject's inherent class properties
class Movable extends GameObject {
    constructor(x,y, type) {
        super(x,y, type)
    }
}

//this movable object can be moved on the screen
moveTo(x, y) {
    this.x = x;
    this.y = y;
}
}

//this is a specific class that extends the Movable class, so it can take :
class Hero extends Movable {
    constructor(x,y) {
        super(x,y, 'Hero')
    }
}

//this class, on the other hand, only inherits the GameObject properties
class Tree extends GameObject {
    constructor(x,y) {
        super(x,y, 'Tree')
    }
}

//a hero can move...
const hero = new Hero();
hero.moveTo(5,5);

//but a tree cannot
const tree = new Tree();

```

- ✓ Take a few minutes to re-envision a Pac-Man hero (Inky, Pinky or Blinky, for example) and how it would be written in JavaScript.

Composition

A different way of handling object inheritance is by using *Composition*. Then, objects express their behavior like this:

javascript

```
//create a constant gameObject
const gameObject = {
  x: 0,
  y: 0,
  type: ''
};

//...and a constant movable
const movable = {
  moveTo(x, y) {
    this.x = x;
    this.y = y;
  }
}

//then the constant movableObject is composed of the gameObject and movable
const movableObject = {...gameObject, ...movable};

//then create a function to create a new Hero who inherits the movableObject
function createHero(x, y) {
  return {
    ...movableObject,
    x,
    y,
    type: 'Hero'
  }
}

//...and a static object that inherits only the gameObject properties
function createStatic(x, y, type) {
  return {
    ...gameObject
    x,
    y,
    type
  }
}

//create the hero and move it
const hero = createHero(10,10);
hero.moveTo(5,5);
```

```
//and create a static tree which only stands around  
const tree = createStatic(0,0, 'Tree');
```

Which pattern should I use?

It's up to you which pattern you choose. JavaScript supports both these paradigms.

--

Another pattern common in game development addresses the problem of handling the game's user experience and performance.

Pub/sub pattern

✅ Pub/Sub stands for 'publish-subscribe'

This pattern addresses the idea that the disparate parts of your application shouldn't know about one another. Why is that? It makes it a lot easier to see what's going on in general if various parts are separated. It also makes it easier to suddenly change behavior if you need to. How do we accomplish this? We do this by establishing some concepts:

- **message:** A message is usually a text string accompanied by an optional payload (a piece of data that clarifies what the message is about). A typical message in a game can be `KEY_PRESSED_ENTER` .
- **publisher:** This element *publishes* a message and sends it out to all subscribers.
- **subscriber:** This element *listens* to specific messages and carries out some task as the result of receiving this message, such as firing a laser.

The implementation is quite small in size but it's a very powerful pattern. Here's how it can be implemented:

javascript

```
//set up an EventEmitter class that contains listeners  
class EventEmitter {  
  constructor() {  
    this.listeners = {};  
  }  
  //when a message is received, let the listener to handle its payload  
  on(message, listener) {  
    if (!this.listeners[message]) {  
      this.listeners[message] = [];  
    }  
    this.listeners[message].push(listener);  
  }  
}
```

```

    }
    //when a message is sent, send it to a listener with some payload
    emit(message, payload = null) {
      if (this.listeners[message]) {
        this.listeners[message].forEach(l => l(message, payload))
      }
    }
  }
}

```

To use the above code we can create a very small implementation:

javascript

```

//set up a message structure
const Messages = {
  HERO_MOVE_LEFT: 'HERO_MOVE_LEFT'
};
//invoke the eventEmitter you set up above
const eventEmitter = new EventEmitter();
//set up a hero
const hero = createHero(0,0);
//let the eventEmitter know to watch for messages pertaining to the hero m
eventEmitter.on(Messages.HERO_MOVE_LEFT, () => {
  hero.move(5,0);
});

//set up the window to listen for the keyup event, specifically if the left
window.addEventListener('keyup', (evt) => {
  if (evt.key === 'ArrowLeft') {
    eventEmitter.emit(Messages.HERO_MOVE_LEFT)
  }
});

```

Above we connect a keyboard event, `ArrowLeft` and send the `HERO_MOVE_LEFT` message. We listen to that message and move the `hero` as a result. The strength with this pattern is that the event listener and the hero don't know about each other. You can remap the `ArrowLeft` to the `A` key. Additionally it would be possible to do something completely different on `ArrowLeft` by making a few edits to the eventEmitter's `on` function:

javascript

```

eventEmitter.on(Messages.HERO_MOVE_LEFT, () => {
  hero.move(5,0);
});

```

As things gets more complicated when your game grows, this pattern stays the same in complexity and your code stays clean. It's really recommended to adopt this pattern.

Challenge

Think about how the pub-sub pattern can enhance a game. Which parts should emit events, and how should the game react to them? Now's your chance to get creative, thinking of a new game and how its parts might behave.

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Learn more about Pub/Sub by [reading about it](#).

Assignment

[Mock up a game](#)

Build a Space Game Part 2: Draw Hero and Monsters to Canvas

Pre-Lecture Quiz

[Pre-lecture quiz](#)

The Canvas

The canvas is an HTML element that by default has no content; it's a blank slate. You need to add to it by drawing on it.

✓ Read [more about the Canvas API](#) on MDN.

Here's how it's typically declared, as part of the page's body:

html

```
<canvas id="myCanvas" width="200" height="100"></canvas>
```

Above we are setting the `id`, `width` and `height`.

- `id`: set this so you can obtain a reference when you need to interact with it.
- `width`: this is the width of the element.
- `height`: this is the height of the element.

Drawing simple geometry

The Canvas is using a cartesian coordinate system to draw things. Thus it uses an x-axis and y-axis to express where something is located. The location `0,0` is the top left position and the bottom right is what you said to be the `WIDTH` and `HEIGHT` of the canvas.

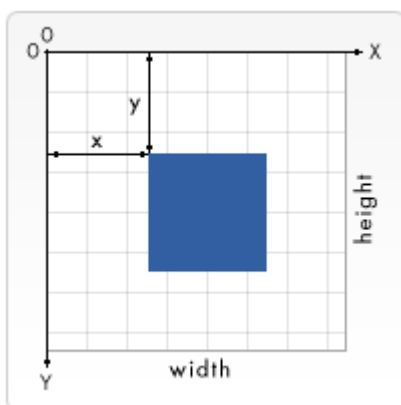


Image from [MDN](#)

To draw on the canvas element you will need to go through the following steps:

1. **Get a reference** to the Canvas element.
2. **Get a reference** on the Context element that sits on the canvas element.
3. **Perform a drawing operation** using the context element.

Code for the above steps usually looks like so:

javascript

```
// draws a red rectangle
//1. get the canvas reference
canvas = document.getElementById("myCanvas");

//2. set the context to 2D to draw basic shapes
ctx = canvas.getContext("2d");

//3. fill it with the color red
ctx.fillStyle = 'red';

//4. and draw a rectangle with these parameters, setting location and size
ctx.fillRect(0,0, 200, 200) // x,y,width, height
```

✅ The Canvas API mostly focuses on 2D shapes, but you can also draw 3D elements to a web site; for this, you might use the [WebGL API](#).

You can draw all sorts of things with the Canvas API like:

- **Geometrical shapes**, we've already showed how to draw a rectangle, but there is much more you can draw.
- **Text**, you can draw a text with any font and color you wish.
- **Images**, you can draw an image based off of an image asset like a .jpg or .png for example.

✅ Try it! You know how to draw a rectangle, can you draw a circle to a page? Take a look at some interesting Canvas drawings on CodePen. Here's a [particularly impressive example](#).

Load and draw an image asset

You load an image asset by creating an `Image` object and set its `src` property. Then you listen to the `load` event to know when it's ready to be used. The code looks like this:

Load asset


```
const img = new Image();
img.src = 'path/to/my/image.png';
img.onload = () => {
  // image loaded and ready to be used
}
```

Load asset pattern

It's recommended to wrap the above in a construct like so, so it's easier to use and you only try to manipulate it when it's fully loaded:

javascript

```
function loadAsset(path) {
  return new Promise((resolve) => {
    const img = new Image();
    img.src = path;
    img.onload = () => {
      // image loaded and ready to be used
      resolve(img);
    }
  })
}
```

// use like so

```
async function run() {
  const heroImg = await loadAsset('hero.png')
  const monsterImg = await loadAsset('monster.png')
}
```

To draw game assets to a screen, your code would look like this:

javascript

```
async function run() {
  const heroImg = await loadAsset('hero.png')
  const monsterImg = await loadAsset('monster.png')

  canvas = document.getElementById("myCanvas");
  ctx = canvas.getContext("2d");
  ctx.drawImage(heroImg, canvas.width/2, canvas.height/2);
  ctx.drawImage(monsterImg, 0, 0);
}
```

Now it's time to start building your game

What to build

You will build a web page with a Canvas element. It should render a black screen `1024*768` . We've provided you with two images:

- Hero ship



- 5*5 monster



Recommended steps to start development

Locate the files that have been created for you in the `your-work` sub folder. It should contain the following:

```
bash
-| assets
  -| enemyShip.png
  -| player.png
-| index.html
-| app.js
-| package.json
```

Open the copy of this folder in Visual Studio Code. You need to have a local development environment setup, preferably with Visual Studio Code with NPM and Node installed. If you don't have `npm` set up on your computer, [here's how to do that](#).

Start your project by navigating to the `your_work` folder:

```
bash
cd your-work
npm start
```

The above will start a HTTP Server on address `http://localhost:5000` . Open up a browser and input that address. It's a blank page right now, but that will change

Note: to see changes on your screen, refresh your browser.

Add code

Add the needed code to `your-work/app.js` to solve the below

1. Draw a canvas with black background

tip: add two lines under the appropriate TODO in `/app.js` , setting the `ctx` element to be black and the top/left coordinates to be at 0,0 and the height and width to equal that of the canvas.

2. Load textures

tip: add the player and enemy images using `await loadImage` and passing in the image path. You won't see them on the screen yet!

3. Draw hero in the center of the screen in the bottom half

tip: use the `drawImage` API to draw `heroImg` to the screen, setting `canvas.width / 2 - 45` and `canvas.height - canvas.height / 4` ;

4. Draw 5*5 monsters

tip: Now you can uncomment the code to draw enemies on the screen. Next, go to the `createEnemies` function and build it out.

First, set up some constants:

javascript

```
const MONSTER_TOTAL = 5;
const MONSTER_WIDTH = MONSTER_TOTAL * 98;
const START_X = (canvas.width - MONSTER_WIDTH) / 2;
const STOP_X = START_X + MONSTER_WIDTH;
```

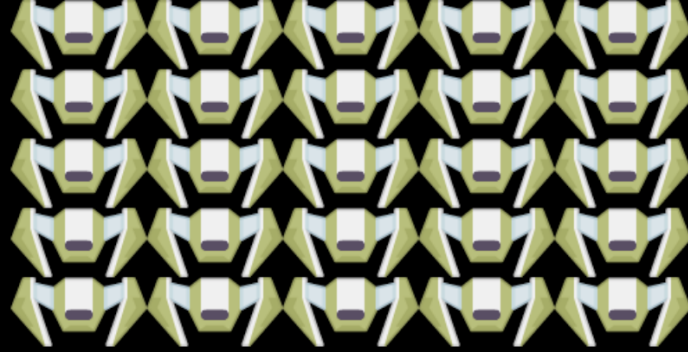
then, create a loop to draw the array of monsters onto the screen:

javascript

```
for (let x = START_X; x < STOP_X; x += 98) {
  for (let y = 0; y < 50 * 5; y += 50) {
    ctx.drawImage(enemyImg, x, y);
  }
}
```

Result

The finished result should look like so:



Solution

Please try solving it yourself first but if you get stuck, have a look at a [solution](#)

Challenge

You've learned about drawing with the 2D-focused Canvas API; take a look at the [WebGL API](#), and try to draw a 3D object.

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Learn more about the Canvas API by [reading about it](#).

Assignment

[Play with the Canvas API](#)

Build a Space Game Part 3: Adding Motion

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Games aren't much fun until you have aliens running around on screen! In this game, we will make use of two types of movements:

- **Keyboard/Mouse movement:** when the user interacts with the keyboard or mouse to move an object on the screen.
- **Game induced movement:** when the game moves an object with a certain time interval.

So how do we move things on a screen? It's all about cartesian coordinates: we change the location (x,y) of the object and then redraw the screen.

Typically you need the following steps to accomplish *movement* on a screen:

1. **Set a new location** for an object; this is needed to perceive the object as having moved.
2. **Clear the screen**, the screen needs to be cleared in between draws. We can clear it by drawing a rectangle that we fill with a background color.
3. **Redraw object** at new location. By doing this we finally accomplish moving the object from one location to the other.

Here's what it can look like in code:

javascript

```
//set the hero's location  
hero.x += 5;
```

```
// clear the rectangle that hosts the hero
ctx.clearRect(0, 0, canvas.width, canvas.height);
// redraw the game background and hero
ctx.fillRect(0, 0, canvas.width, canvas.height)
ctx.fillStyle = "black";
ctx.drawImage(heroImg, hero.x, hero.y);
```

✅ Can you think of a reason why redrawing your hero many frames per second might accrue performance costs? Read about [alternatives to this pattern](#).

Handle keyboard events

You handle events by attaching specific events to code. Keyboard events are triggered on the whole window whereas mouse events like a `click` can be connected to clicking a specific element. We will use keyboard events throughout this project.

To handle an event you need to use the window's `addEventListener()` method and provide it with two input parameters. The first parameter is the name of the event, for example `keyup`. The second parameter is the function that should be invoked as a result of the event taking place.

Here's an example:

javascript

```
window.addEventListener('keyup', (evt) => {
  // `evt.key` = string representation of the key
  if (evt.key === 'ArrowUp') {
    // do something
  }
})
```

For key events there are two properties on the event you can use to see what key was pressed:

- `key`, this is a string representation of the pressed key, for example `ArrowUp`
- `keyCode`, this is a number representation, for example `37`, corresponds to `ArrowLeft`.

✅ Key event manipulation is useful outside of game development. What other uses can you think of for this technique?

Special keys: a caveat

There are some *special* keys that affect the window. That means that if you are listening to a `keyup` event and you use these special keys to move your hero it will also perform horizontal scrolling. For that reason you might want to *shut-off* this built-in browser behavior as you build out your game. You need code like this:

javascript

```
let onKeyDown = function (e) {
  console.log(e.keyCode);
  switch (e.keyCode) {
    case 37:
    case 39:
    case 38:
    case 40: // Arrow keys
    case 32:
      e.preventDefault();
      break; // Space
    default:
      break; // do not block other keys
  }
};

window.addEventListener('keydown', onKeyDown);
```

The above code will ensure that arrow-keys and the space key have their *default* behavior shut off. The *shut-off* mechanism happens when we call `e.preventDefault()`.

Game induced movement

We can make things move by themselves by using timers such as the `setTimeout()` or `setInterval()` function that update the location of the object on each tick, or time interval. Here's what that can look like:

javascript

```
let id = setInterval(() => {
  //move the enemy on the y axis
  enemy.y += 10;
})
```

The game loop

The game loop is a concept that is essentially a function that is invoked at regular intervals. It's called the game loop as everything that should be visible to the user is drawn into the loop. The game loop makes use of all the game objects that are part of the game, drawing all of them unless for some reason shouldn't be part of the game any more. For example if an object is an enemy that was hit by a laser and blows up, it's no longer part of the current game loop (you'll learn more on this in subsequent lessons).

Here's what a game loop can typically look like, expressed in code:

javascript

```
let gameLoopId = setInterval(() =>
  function gameLoop() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.fillStyle = "black";
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    drawHero();
    drawEnemies();
    drawStaticObjects();
  }, 200);
```

The above loop is invoked every 200 milliseconds to redraw the canvas. You have the ability to choose the best interval that makes sense for your game.

Continuing the Space Game

You will take the existing code and extend it. Either start with the code that you completed during part I or use the code in [Part II- starter](#).

- **Moving the hero:** you will add code to ensure you can move the hero using the arrow keys.
- **Move enemies:** you will also need to add code to ensure the enemies move from top to bottom at a given rate.

Recommended steps

Locate the files that have been created for you in the `your-work` sub folder. It should contain the following:

```
-| assets
  -| enemyShip.png
  -| player.png
-| index.html
-| app.js
-| package.json
```

You start your project the `your_work` folder by typing:

```
cd your-work
npm start
```

The above will start a HTTP Server on address `http://localhost:5000` . Open up a browser and input that address, right now it should render the hero and all the enemies; nothing is moving - yet!

Add code

1. **Add dedicated objects** for hero and enemy and game object , they should have `x` and `y` properties. (Remember the portion on [Inheritance or composition](#)).

HINT game object should be the one with `x` and `y` and the ability to draw itself to a canvas.

tip: start by adding a new `GameObject` class with its constructor delineated as below, and then draw it to the canvas:

```
class GameObject {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.dead = false;
    this.type = "";
    this.width = 0;
    this.height = 0;
    this.img = undefined;
  }
}
```

```

    }

    draw(ctx) {
      ctx.drawImage(this.img, this.x, this.y, this.width, this.height);
    }
  }
}

```

Now, extend this GameObject to create the Hero and Enemy.

javascript

```

class Hero extends GameObject {
  constructor(x, y) {
    ...it needs an x, y, type, and speed
  }
}

```

javascript

```

class Enemy extends GameObject {
  constructor(x, y) {
    super(x, y);
    (this.width = 98), (this.height = 50);
    this.type = "Enemy";
    let id = setInterval(() => {
      if (this.y < canvas.height - this.height) {
        this.y += 5;
      } else {
        console.log('Stopped at', this.y)
        clearInterval(id);
      }
    }, 300)
  }
}

```

2. Add key-event handlers to handle key navigation (move hero up/down left/right)

REMEMBER it's a cartesian system, top-left is $0,0$. Also remember to add code to stop *default behavior*

tip: create your onKeyDown function and attach it to the window:

```

let onKeyDown = function (e) {
  console.log(e.keyCode);
  ...add the code from the lesson above to stop default behavior
}
};

window.addEventListener("keydown", onKeyDown);

```

Check your browser console at this point, and watch the keystrokes being logged.

3. **Implement** the Pub sub pattern, this will keep your code clean as you follow the remaining parts.

To do this last part, you can:

1. **Add an event listener** on the window:

```

window.addEventListener("keyup", (evt) => {
  if (evt.key === "ArrowUp") {
    eventEmitter.emit(Messages.KEY_EVENT_UP);
  } else if (evt.key === "ArrowDown") {
    eventEmitter.emit(Messages.KEY_EVENT_DOWN);
  } else if (evt.key === "ArrowLeft") {
    eventEmitter.emit(Messages.KEY_EVENT_LEFT);
  } else if (evt.key === "ArrowRight") {
    eventEmitter.emit(Messages.KEY_EVENT_RIGHT);
  }
});

```

2. **Create an EventEmitter class** to publish and subscribe to messages:

```

class EventEmitter {
  constructor() {
    this.listeners = {};
  }

  on(message, listener) {
    if (!this.listeners[message]) {
      this.listeners[message] = [];
    }
    this.listeners[message].push(listener);
  }
}

```

```

emit(message, payload = null) {
  if (this.listeners[message]) {
    this.listeners[message].forEach((l) => l(message, payload));
  }
}
}
}

```

3. Add constants and set up the EventEmitter:

javascript

```

const Messages = {
  KEY_EVENT_UP: "KEY_EVENT_UP",
  KEY_EVENT_DOWN: "KEY_EVENT_DOWN",
  KEY_EVENT_LEFT: "KEY_EVENT_LEFT",
  KEY_EVENT_RIGHT: "KEY_EVENT_RIGHT",
};

let heroImg,
    enemyImg,
    laserImg,
    canvas, ctx,
    gameObjects = [],
    hero,
    eventEmitter = new EventEmitter();

```

4. Initialize the game

javascript

```

function initGame() {
  gameObjects = [];
  createEnemies();
  createHero();

  eventEmitter.on(Messages.KEY_EVENT_UP, () => {
    hero.y -= 5;
  });

  eventEmitter.on(Messages.KEY_EVENT_DOWN, () => {
    hero.y += 5;
  });

  eventEmitter.on(Messages.KEY_EVENT_LEFT, () => {
    hero.x -= 5;
  });
}

```

```
});
```

```
    eventEmitter.on(Messages.KEY_EVENT_RIGHT, () => {  
      hero.x += 5;  
    });  
  }  
}
```

4. Setup the game loop

Refactor the `window.onload` function to initialize the game and set up a game loop on a good interval. You'll also add a laser beam:

javascript

```
window.onload = async () => {  
  canvas = document.getElementById("canvas");  
  ctx = canvas.getContext("2d");  
  heroImg = await loadTexture("assets/player.png");  
  enemyImg = await loadTexture("assets/enemyShip.png");  
  laserImg = await loadTexture("assets/laserRed.png");  
  
  initGame();  
  let gameLoopId = setInterval(() => {  
    ctx.clearRect(0, 0, canvas.width, canvas.height);  
    ctx.fillStyle = "black";  
    ctx.fillRect(0, 0, canvas.width, canvas.height);  
    drawGameObjects(ctx);  
  }, 100)  
};
```

5. Add code to move enemies at a certain interval

Refactor the `createEnemies()` function to create the enemies and push them into the new `gameObjects` class:

javascript

```
function createEnemies() {  
  const MONSTER_TOTAL = 5;  
  const MONSTER_WIDTH = MONSTER_TOTAL * 98;  
  const START_X = (canvas.width - MONSTER_WIDTH) / 2;  
  const STOP_X = START_X + MONSTER_WIDTH;  
  
  for (let x = START_X; x < STOP_X; x += 98) {  
    for (let y = 0; y < 50 * 5; y += 50) {
```

```
    const enemy = new Enemy(x, y);
    enemy.img = enemyImg;
    gameObjects.push(enemy);
  }
}
```

and add a `createHero()` function to do a similar process for the hero.

javascript

```
function createHero() {
  hero = new Hero(
    canvas.width / 2 - 45,
    canvas.height - canvas.height / 4
  );
  hero.img = heroImg;
  gameObjects.push(hero);
}
```

and finally, add a `drawGameObjects()` function to start the drawing:

javascript

```
function drawGameObjects(ctx) {
  gameObjects.forEach(go => go.draw(ctx));
}
```

Your enemies should start advancing on your hero spaceship!

Challenge

As you can see, your code can turn into 'spaghetti code' when you start adding functions and variables and classes. How can you better organize your code so that it is more readable? Sketch out a system to organize your code, even if it still resides in one file.

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

While we're writing our game without using frameworks, there are many JavaScript-based canvas frameworks for game development. Take some time to do some [reading about these](#).

Assignment

[Comment your code](#)

Build a Space Game Part 4: Adding A Laser and Detect Collisions

Pre-Lecture Quiz

[Pre-lecture quiz](#)

In this lesson you will learn how to shoot lasers with JavaScript! We will add two things to our game:

- **A laser:** this laser is shot from your heroes ship and vertically upwards
- **Collision detection,** as part of implementing the ability to *shoot* we will also add some nice game rules:
 - **Laser hits enemy:** Enemy dies if hit by a laser
 - **Laser hits top screen:** A laser is destroyed if hitting the top part of the screen
 - **Enemy and hero collision:** An enemy and the hero are destroyed if hitting each other
 - **Enemy hits bottom of the screen:** An enemy and a hero are destroyed if the enemy hits the bottom of the screen

In short, you -- *the hero* -- need to hit all enemies with a laser before they manage to move to the bottom of the screen.

Do a little research on the very first computer game ever written. What was its functionality?

Let's be heroic together!

Collision detection

How do we do collision detection? We need to think of our game objects as rectangles moving about. Why is that you might ask? Well, the image used to draw a game object is a rectangle: it has an `x`, `y`, `width` and `height`.

If two rectangles, i.e a hero and enemy *intersect*, you have a collision. What should happen then is up to the rules of the game. To implement collision detection you therefore need the following:

1. A way to get a rectangle representation of a game object, something like this:

javascript

```
rectFromGameObject() {  
  return {  
    top: this.y,  
    left: this.x,  
    bottom: this.y + this.height,  
    right: this.x + this.width  
  }  
}
```

2. A comparison function, this function can look like this:

javascript

```
function intersectRect(r1, r2) {  
  return !(r2.left > r1.right ||  
    r2.right < r1.left ||  
    r2.top > r1.bottom ||  
    r2.bottom < r1.top);  
}
```

How do we destroy things

To destroy things in a game you need to let the game know it should no longer paint this item in the game loop that triggers on a certain interval. A way to do this is to mark a game object as *dead* when something happens, like so:

javascript

```
// collision happened  
enemy.dead = true
```

Then you can proceed to sort out *dead* objects before repainting the screen, like so:

javascript

```
gameObjects = gameObject.filter(go => !go.dead);
```

How do we fire a laser

Firing a laser translates to responding to a key-event and creating an object that moves in a certain direction. We therefore need to carry out the following steps:

1. **Create a laser object:** from the top of our hero's ship, that upon creation starts moving upwards towards the screen top.
2. **Attach code to a key event:** we need to choose a key on the keyboard that represents the player shooting the laser.
3. **Create a game object that looks like a laser** when the key is pressed.

Cooldown on our laser

The laser needs to fire every time you press a key, like *space* for example. To prevent the game producing way too many lasers in a short time we need to fix this. The fix is by implementing a so called *cooldown*, a timer, that ensures that a laser can only be fired so often. You can implement that in the following way:

javascript

```
class Cooldown {
  constructor(time) {
    this.cool = false;
    setTimeout(() => {
      this.cool = true;
    }, time)
  }
}

class Weapon {
  constructor {
  }
  fire() {
    if (!this.cooldown || this.cooldown.cool) {
      // produce a laser
    }
  }
}
```

```
    this.cooldown = new Cooldown(500);
  } else {
    // do nothing - it hasn't cooled down yet.
  }
}
}
```

✔ Refer to lesson 1 in the space game series to remind yourself about *cooldowns*.

What to build

You will take the existing code (which you should have cleaned up and refactored) from the previous lesson, and extend it. Either start with the code from part II or use the code at [Part III- starter](#).

tip: the laser that you'll work with is already in your assets folder and referenced by your code

- **Add collision detection**, when a laser collides with something the following rules should apply:
 1. **Laser hits enemy**: enemy dies if hit by a laser
 2. **Laser hits top screen**: A laser is destroyed if it hits the top part of our screen
 3. **Enemy and hero collision**: an enemy and the hero is destroyed if hitting each other
 4. **Enemy hits bottom of the screen**: An enemy and a hero is destroyed if the enemy hits the bottom of the screen

Recommended steps

Locate the files that have been created for you in the `your-work` sub folder. It should contain the following:

```
bash
-| assets
  -| enemyShip.png
  -| player.png
  -| laserRed.png
-| index.html
-| app.js
-| package.json
```

You start your project the `your_work` folder by typing:

bash

```
cd your-work
npm start
```

The above will start a HTTP Server on address `http://localhost:5000` . Open up a browser and input that address, right now it should render the hero and all the enemies, nothing is moving - yet :).

Add code

1. **Setup a rectangle representation of your game object, to handle collision** The below code allows you to get a rectangle representation of a `GameObject` . Edit your `GameObject` class to extend it:

javascript

```
rectFromGameObject() {
  return {
    top: this.y,
    left: this.x,
    bottom: this.y + this.height,
    right: this.x + this.width,
  };
}
```

2. **Add code that checks collision** This will be a new function that tests whether two rectangles intersect:

javascript

```
function intersectRect(r1, r2) {
  return !(
    r2.left > r1.right ||
    r2.right < r1.left ||
    r2.top > r1.bottom ||
    r2.bottom < r1.top
  );
}
```

3. **Add laser firing capability**

1. **Add key-event message.** The `space` key should create a laser just above the hero ship. Add three constants in the `Messages` object:

javascript

```
KEY_EVENT_SPACE: "KEY_EVENT_SPACE",  
COLLISION_ENEMY_LASER: "COLLISION_ENEMY_LASER",  
COLLISION_ENEMY_HERO: "COLLISION_ENEMY_HERO",
```

2. **Handle space key.** Edit the `window.addEventListener` `keyup` function to handle spaces:

javascript

```
    } else if(evt.keyCode === 32) {  
        eventEmitter.emit(Messages.KEY_EVENT_SPACE);  
    }
```

3. **Add listeners.** Edit the `initGame()` function to ensure that hero can fire when the space bar is hit:

javascript

```
eventEmitter.on(Messages.KEY_EVENT_SPACE, () => {  
    if (hero.canFire()) {  
        hero.fire();  
    }  
})
```

and add a new `eventEmitter.on()` function to ensure behavior when an enemy collides with a laser:

javascript

```
eventEmitter.on(Messages.COLLISION_ENEMY_LASER, (_, { first, second }  
    first.dead = true;  
    second.dead = true;  
})
```

4. **Move object,** Ensure the laser moves to the top of the screen gradually. You'll create a new `Laser` class that extends `GameObject`, as you've done before:

javascript

```
class Laser extends GameObject {  
    constructor(x, y) {  
        super(x,y);  
        (this.width = 9), (this.height = 33);  
        this.type = 'Laser';  
        this.img = laserImg;  
    }  
}
```

```

    let id = setInterval(() => {
      if (this.y > 0) {
        this.y -= 15;
      } else {
        this.dead = true;
        clearInterval(id);
      }
    }, 100)
  }
}

```

5. **Handle collisions**, Implement collision rules for the laser. Add an `updateGameObjects()` function that tests colliding objects for hits

javascript

```

function updateGameObjects() {
  const enemies = gameObjects.filter(go => go.type === 'Enemy');
  const lasers = gameObjects.filter((go) => go.type === "Laser");
  // laser hit something
  lasers.forEach((l) => {
    enemies.forEach((m) => {
      if (intersectRect(l.rectFromGameObject(), m.rectFromGameObject(
        emitter.emit(Messages.COLLISION_ENEMY_LASER, {
          first: l,
          second: m,
        }));
    }
  });
});

gameObjects = gameObjects.filter(go => !go.dead);
}

```

Make sure to add `updateGameObjects()` into your game loop in `window.onload` .

6. **Implement cooldown** on the laser, so it can only be fired so often.

Finally, edit the Hero class so that it can cooldown:

javascript

```

class Hero extends GameObject {
  constructor(x, y) {
    super(x, y);
    (this.width = 99), (this.height = 75);
  }
}

```

```
    this.type = "Hero";
    this.speed = { x: 0, y: 0 };
    this.cooldown = 0;
  }
  fire() {
    gameObjects.push(new Laser(this.x + 45, this.y - 10));
    this.cooldown = 500;

    let id = setInterval(() => {
      if (this.cooldown > 0) {
        this.cooldown -= 100;
      } else {
        clearInterval(id);
      }
    }, 200);
  }
  canFire() {
    return this.cooldown === 0;
  }
}
```

At this point, your game has some functionality! You can navigate with your arrow keys, fire a laser with your space bar, and enemies disappear when you hit them. Well done!

Challenge

Add an explosion! Take a look at the game assets in [the Space Art repo](#) and try to add an explosion when the laser hits an alien

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Experiment with the intervals in your game thus far. What happens when you change them? Read more about [JavaScript timing events](#).

Assignment

[Explore collisions](#)

Build a Space Game Part 5: Scoring and Lives

Pre-Lecture Quiz

[Pre-lecture quiz](#)

In this lesson, you'll learn how to add scoring to a game and calculate lives.

Draw text on the screen

To be able to display a game score on the screen, you'll need to know how to place text on the screen. The answer is using the `fillText()` method on the canvas object. You can also control other aspects like what font to use, the color of the text and even its alignment (left, right, center).

Below is some code drawing some text on the screen.

javascript

```
ctx.font = "30px Arial";
ctx.fillStyle = "red";
ctx.textAlign = "right";
ctx.fillText("show this on the screen", 0, 0);
```


✓ Read more about [how to add text to a canvas](#), and feel free to make yours look fancier!

Life, as a game concept

The concept of having a life in a game is only a number. In the context of a space game it's common to assign a set of lives that get deducted one by one when your ship takes damage. It's nice if you can show a graphical representation of this like miniships or hearts instead of a number.

What to build

Let's add the following to your game:

- **Game score:** For every enemy ship that is destroyed, the hero should be awarded some points, we suggest a 100 points per ship. The game score should be shown in the bottom left.
- **Life:** Your ship has three lives. You lose a life every time an enemy ship collides with you. A life score should be displayed at the bottom right and be made out of the following graphic .

Recommended steps

Locate the files that have been created for you in the `your-work` sub folder. It should contain the following:

```
-| assets
  -| enemyShip.png
  -| player.png
  -| laserRed.png
-| index.html
-| app.js
-| package.json
```

bash

You start your project the `your_work` folder by typing:

```
cd your-work
npm start
```

bash

The above will start a HTTP Server on address `http://localhost:5000` . Open up a browser and input that address, right now it should render the hero and all the enemies, and as you hit your left and right arrows, the hero moves and can shoot down enemies.

Add code

1. **Copy over the needed assets** from the `solution/assets/` folder into your-work folder; you will add a `life.png` asset. Add the `lifeImg` to the `window.onload` function:

javascript

```
lifeImg = await loadTexture("assets/life.png");
```

2. Add the `lifeImg` to the list of assets:

javascript

```
let heroImg,  
...  
lifeImg,  
...  
eventEmitter = new EventEmitter();
```

3. **Add variables.** Add code that represents your total score (0) and lives left (3), display these scores on a screen.
4. **Extend `updateGameObjects()` function.** Extend the `updateGameObjects()` function to handle enemy collisions:

javascript

```
enemies.forEach(enemy => {  
    const heroRect = hero.rectFromGameObject();  
    if (intersectRect(heroRect, enemy.rectFromGameObject())) {  
        eventEmitter.emit(Messages.COLLISION_ENEMY_HERO, { enemy });  
    }  
})
```

5. **Add life and points .**

1. **Initialize variables.** Under `this.cooldown = 0` in the `Hero` class, set life and points:

javascript

```
this.life = 3;  
this.points = 0;
```

2. **Draw variables on screen.** Draw these values to screen:

```

function drawLife() {
  // TODO, 35, 27
  const START_POS = canvas.width - 180;
  for(let i=0; i < hero.life; i++ ) {
    ctx.drawImage(
      lifeImg,
      START_POS + (45 * (i+1) ),
      canvas.height - 37);
  }
}

function drawPoints() {
  ctx.font = "30px Arial";
  ctx.fillStyle = "red";
  ctx.textAlign = "left";
  drawText("Points: " + hero.points, 10, canvas.height-20);
}

function drawText(message, x, y) {
  ctx.fillText(message, x, y);
}

```

3. **Add methods to Game loop.** Make sure you add these functions to your window.onload

function under updateGameObjects() :

javascript

```

drawPoints();
drawLife();

```

6. **Implement game rules.** Implement the following game rules:

1. **For every hero and enemy collision,** deduct a life.

Extend the Hero class to do this deduction:

javascript

```

decrementLife() {
  this.life--;
  if (this.life === 0) {
    this.dead = true;
  }
}

```

2. For every laser that hits an enemy, increase game score with a 100 points.

Extend the Hero class to do this increment:

javascript

```
incrementPoints() {  
  this.points += 100;  
}
```

Add these functions to your Collision Event Emitters:

javascript

```
eventEmitter.on(Messages.COLLISION_ENEMY_LASER, (_, { first, second }  
  first.dead = true;  
  second.dead = true;  
  hero.incrementPoints();  
)  
  
eventEmitter.on(Messages.COLLISION_ENEMY_HERO, (_, { enemy }) => {  
  enemy.dead = true;  
  hero.decrementLife();  
});
```

✅ Do a little research to discover other games that are created using JavaScript/Canvas. What are their common traits?

By the end of this work, you should see the small 'life' ships at the bottom right, points at the bottom left, and you should see your life count decrement as you collide with enemies and your points increment when you shoot enemies. Well done! Your game is almost complete.

Challenge

Your code is almost complete. Can you envision your next steps?

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Research some ways that you can increment and decrement game scores and lives. There are some interesting game engines like [PlayFab](#). How could using one of these would enhance your game?

Assignment

[Build a Scoring Game](#)

Build a Space Game Part 6: End and Restart

Pre-Lecture Quiz

[Pre-lecture quiz](#)

There are different ways to express an *end condition* in a game. It's up to you as the creator of the game to say why the game has ended. Here are some reasons, if we assume we are talking about the space game you have been building so far:

- **N Enemy ships have been destroyed:** It's quite common if you divide up a game into different levels that you need to destroy N Enemy ships to complete a level
- **Your ship has been destroyed:** There are definitely games where you lose the game if your ship is destroyed. Another common approach is that you have the concept of lives. Every time a your ship is destroyed it deducts a life. Once all lives have been lost then you lose the game.
- **You've collected N points:** Another common end condition is for you to collect points. How you get points is up to you but it's quite common to assign points to various activities like destroying an enemy ship or maybe collect items that items *drop* when they are destroyed.
- **Complete a level:** This might involve several conditions such as X enemy ships destroyed, Y points collected or maybe that a specific item has been collected.

Restarting

If people enjoy your game they are likely to want to replay it. Once the game ends for whatever reason you should offer an alternative to restart.

✔ Think a bit about under what conditions you find a game ends, and then how you are prompted to restart

What to build

You will be adding these rules to your game:

1. **Winning the game.** Once all enemy ships have been destroyed, you win the game. Additionally display some kind of victory message.
2. **Restart.** Once all your lives are lost or the game is won, you should offer a way to restart the game. Remember! You will need to reinitialize the game and the previous game state should be cleared.

Recommended steps

Locate the files that have been created for you in the `your-work` sub folder. It should contain the following:

```
bash
-| assets
  -| enemyShip.png
  -| player.png
  -| laserRed.png
  -| life.png
-| index.html
-| app.js
-| package.json
```

You start your project the `your_work` folder by typing:

```
bash
cd your-work
npm start
```

The above will start a HTTP Server on address `http://localhost:5000` . Open up a browser and input that address. Your game should be in a playable state.

tip: to avoid warnings in Visual Studio Code, edit the `window.onload` function to call `gameLoopId` as is (without `let`), and declare the `gameLoopId` at the top of the file, independently: `let gameLoopId;`

Add code

1. **Track end condition.** Add code that keeps track of the number of enemies, or if the hero ship has been destroyed by adding these two functions:

javascript

```
function isHeroDead() {
  return hero.life <= 0;
}

function isEnemiesDead() {
  const enemies = gameObjects.filter((go) => go.type === "Enemy" && !go.
  return enemies.length === 0;
}
```

2. **Add logic to message handlers.** Edit the `eventEmitter` to handle these conditions:

javascript

```
eventEmitter.on(Messages.COLLISION_ENEMY_LASER, (_, { first, second }) =
  first.dead = true;
  second.dead = true;
  hero.incrementPoints();

  if (isEnemiesDead()) {
    eventEmitter.emit(Messages.GAME_END_WIN);
  }
});

eventEmitter.on(Messages.COLLISION_ENEMY_HERO, (_, { enemy }) => {
  enemy.dead = true;
  hero.decrementLife();
  if (isHeroDead()) {
    eventEmitter.emit(Messages.GAME_END_LOSS);
    return; // loss before victory
  }
  if (isEnemiesDead()) {
    eventEmitter.emit(Messages.GAME_END_WIN);
  }
});
```

```

    }
  });

  emitter.on(Messages.GAME_END_WIN, () => {
    endGame(true);
  });

  emitter.on(Messages.GAME_END_LOSS, () => {
    endGame(false);
  });

```

3. **Add new message types.** Add these Messages to the constants object:

javascript

```

GAME_END_LOSS: "GAME_END_LOSS",
GAME_END_WIN: "GAME_END_WIN",

```

4. **Add restart code** code that restarts the game at the press of a selected button.

1. **Listen to key press** Enter . Edit your window's eventListener to listen for this press:

javascript

```

else if(evt.key === "Enter") {
  emitter.emit(Messages.KEY_EVENT_ENTER);
}

```

2. **Add restart message.** Add this Message to your Messages constant:

javascript

```

KEY_EVENT_ENTER: "KEY_EVENT_ENTER",

```

5. **Implement game rules.** Implement the following game rules:

1. **Player win condition.** When all enemy ships are destroyed, display a victory message.

1. First, create a `displayMessage()` function:

javascript

```

function displayMessage(message, color = "red") {
  ctx.font = "30px Arial";
  ctx.fillStyle = color;
  ctx.textAlign = "center";
  ctx.fillText(message, canvas.width / 2, canvas.height / 2);
}

```


2. Create an `endGame()` function:

javascript

```
function endGame(win) {
  clearInterval(gameLoopId);

  // set a delay so we are sure any paints have finished
  setTimeout(() => {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.fillStyle = "black";
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    if (win) {
      displayMessage(
        "Victory!!! Pew Pew... - Press [Enter] to start a new game (",
        "green"
      );
    } else {
      displayMessage(
        "You died !!! Press [Enter] to start a new game Captain Pew",
        "red"
      );
    }
  }, 200)
}
```

2. **Restart logic.** When all lives are lost or the player won the game, display that the game can be restarted. Additionally restart the game when the *restart* key is hit (you can decide what key should be mapped to restart).

1. Create the `resetGame()` function:

javascript

```
function resetGame() {
  if (gameLoopId) {
    clearInterval(gameLoopId);
    eventEmitter.clear();
    initGame();
    gameLoopId = setInterval(() => {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.fillStyle = "black";
      ctx.fillRect(0, 0, canvas.width, canvas.height);
      drawPoints();
      drawLife();
      updateGameObjects();
    }, 1000);
  }
}
```

```
        drawGameObjects(ctx);
    }, 100);
}
}
```

3. Add a call to the `eventEmitter` to reset the game in `initGame()` :

```
eventEmitter.on(Messages.KEY_EVENT_ENTER, () => {
    resetGame();
});
```

javascript

4. Add a `clear()` function to the `EventEmitter`:

```
clear() {
    this.listeners = {};
}
```

javascript

👾 🌟 🚀 Congratulations, Captain! Your game is complete! Well done! 🚀 🌟 👾

Challenge

Add a sound! Can you add a sound to enhance your game play, maybe when there's a laser hit, or the hero dies or wins? Have a look at this [sandbox](#) to learn how to play sound using JavaScript

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Your assignment is to create a fresh sample game, so explore some of the interesting games out there to see what type of game you might build.

Assignment

[Build a Sample Game](#)

Build a Banking App Part 1: HTML Templates and Routes in a Web App

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

Since the advent of JavaScript in browsers, websites are becoming more interactive and complex than ever. Web technologies are now commonly used to create fully functional applications that runs directly into a browser that we call [web applications](#). As Web apps are highly interactive, users do not want to wait for a full page reload every time an action is performed. That's why JavaScript is used to update the HTML directly using the DOM, to provide a smoother user experience.

In this lesson, we're going to lay out the foundations to create bank web app, using HTML templates to create multiple screens that can be displayed and updated without having to reload the entire HTML page.

Prerequisite

You need a local web server to test the web app we'll build in this lesson. If don't have one, you can install [Node.js](#) and use the command `npx lite-server` from your project folder. It will create a local web server and open your app in a browser.

Preparation

On your computer, create a folder named `bank` with a file named `index.html` inside it. We'll start from this HTML [boilerplate](#):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Bank App</title>
  </head>
  <body>
    <!-- This is where you'll work -->
  </body>
</html>
```

HTML templates

If you want to create multiples screens for a web page, one solution would be to create one HTML file for every screen you want to display. However, this solution comes with some inconvenience:

- You have to reload the entire HTML when switching screen, which can be slow.
- It's difficult to share data between the different screens.

Another approach is have only one HTML file, and define multiple [HTML templates](#) using the

`<template>` element. A template is a reusable HTML block that is not displayed by the browser, and needs to be instantiated at runtime using JavaScript.

Task

We'll create a bank app with two screens: the login page and the dashboard. First, let's add in the HTML body a placeholder element that we'll use to instantiate the different screens of our app:

html

```
<div id="app">Loading...</div>
```

We're giving it an `id` to make it easier to locate it with JavaScript later.

Tip: since the content of this element will be replaced, we can put in a loading message or indicator that will be shown while the app is loading.

Next, let's add below the HTML template for the login page. For now we'll only put in there a title and a section containing a link that we'll use to perform the navigation.

html

```
<template id="login">
  <h1>Bank App</h1>
  <section>
    <a href="/dashboard">Login</a>
  </section>
</template>
```

Then we'll add another HTML template for the dashboard page. This page will contain different sections:

- A header with a title and a logout link
- The current balance of the bank account
- A list of transactions, displayed in a table

html

```
<template id="dashboard">
  <header>
    <h1>Bank App</h1>
    <a href="/login">Logout</a>
  </header>
  <section>
    Balance: 100$
  </section>
  <section>
    <h2>Transactions</h2>
    <table>
      <thead>
        <tr>
          <th>Date</th>
          <th>Object</th>
          <th>Amount</th>
        </tr>
      </thead>
      <tbody></tbody>
    </table>
  </section>
</template>
```

Tip: when creating HTML templates, if you want to see what it will look like, you can comment out the `<template>` and `</template>` lines by enclosing them with `<!-- -->` .

✓ Why do you think we use `id` attributes on the templates? Could we use something else like classes?

Displaying templates with JavaScript

If you try your current HTML file in a browser, you'll see that it get stuck displaying `Loading...` . That's because we need to add some JavaScript code to instantiate and display the HTML templates.

Instantiating a template is usually done in 3 steps:

1. Retrieve the template element in the DOM, for example using `document.getElementById` .
2. Clone the template element, using `cloneNode` .
3. Attach it to the DOM under a visible element, for example using `appendChild` .

✓ Why do we need to clone the template before attaching it to the DOM? What do you think would happen if we skipped this step?

Task

Create a new file named `app.js` in your project folder and import that file in the `<head>` section of your HTML:

html

```
<script src="app.js" defer></script>
```

Now in `app.js` , we'll create a new function `updateRoute` :

js

```
function updateRoute(templateId) {  
  const template = document.getElementById(templateId);  
  const view = template.content.cloneNode(true);  
  const app = document.getElementById('app');  
  app.innerHTML = '';  
  app.appendChild(view);  
}
```

What we do here is exactly the 3 steps described above. We instantiate the template with the id `templateId`, and put its cloned content within our app placeholder. Note that we need to use `cloneNode(true)` to copy the entire subtree of the template.

Now call this function with one of the template and look at the result.

js

```
updateRoute('login');
```

✔ What's the purpose of this code `app.innerHTML = ''`; ? What happens without it?

Creating routes

When talking about a web app, we call *Routing* the intent to map **URLs** to specific screens that should be displayed. On a web site with multiple HTML files, this is done automatically as the file paths are reflected on the URL. For example, with these files in your project folder:

```
mywebsite/index.html
mywebsite/login.html
mywebsite/admin/index.html
```

If you create a web server with `mywebsite` as the root, the URL mapping will be:

```
https://site.com          --> mywebsite/index.html
https://site.com/login.html --> mywebsite/login.html
https://site.com/admin/   --> mywebsite/admin/index.html
```

However, for our web app we are using a single HTML file containing all the screens so this default behavior won't help us. We have to create this map manually and perform update the displayed template using JavaScript.

Task

We'll use a simple object to implement a map between URL paths and our templates. Add this object at the top of your `app.js` file.

```
const routes = {
  '/login': { templateId: 'login' },
  '/dashboard': { templateId: 'dashboard' },
};
```

Now let's modify a bit the `updateRoute` function. Instead of passing directly the `templateId` as an argument, we want to retrieve it by first looking at the current URL, and then use our map to get the corresponding template ID value. We can use `window.location.pathname` to get only the path section from the URL.

```
function updateRoute() {
  const path = window.location.pathname;
  const route = routes[path];

  const template = document.getElementById(route.templateId);
  const view = template.content.cloneNode(true);
  const app = document.getElementById('app');
  app.innerHTML = '';
  app.appendChild(view);
}
```

Here we mapped the routes we declared to the corresponding template. You can try it that it works correctly by changing the URL manually in your browser.

✅ What happens if you enter an unknown path in the URL? How could we solve this?

Adding navigation

The next step for our app is to add the possibility to navigate between pages without having to change the URL manually. This implies two things:

1. Updating the current URL
2. Updating the displayed template based on the new URL

We already took care of the second part with the `updateRoute` function, so we have to figure out how to update the current URL.

We'll have to use JavaScript and more specifically the `history.pushState` that allows to update the URL and create a new entry in the browsing history, without reloading the HTML.

Note: While the HTML anchor element `<a href>` can be used on its own to create hyperlinks to different URLs, it will make the browser reload the HTML by default. It is necessary to prevent this behavior when handling routing with custom javascript, using the `preventDefault()` function on the click event.

Task

Let's create a new function we can use to navigate in our app:

```
function navigate(path) {
  window.history.pushState({}, path, path);
  updateRoute();
}
```

js

This method first updates the current URL based on the path given, then updates the template. The property `window.location.origin` returns the URL root, allowing us to reconstruct a complete URL from a given path.

Now that we have this function, we can take care of the problem we have if a path does not match any defined route. We'll modify the `updateRoute` function by adding a fallback to one of the existing route if we can't find a match.

```
function updateRoute() {
  const path = window.location.pathname;
  const route = routes[path];

  if (!route) {
    return navigate('/login');
  }

  ...
}
```

js

If a route cannot be found, we'll now redirect to the `login` page.

Now let's create a function to get the URL when a link is clicked, and to prevent the browser's default link behavior:

```
function onLinkClick(event) {  
  event.preventDefault();  
  navigate(event.target.href);  
}
```

Let's complete the navigation system by adding bindings to our *Login* and *Logout* links in the HTML.

html

```
<a href="/dashboard" onclick="onLinkClick()">Login</a>  
...  
<a href="/login" onclick="onLinkClick()">Logout</a>
```

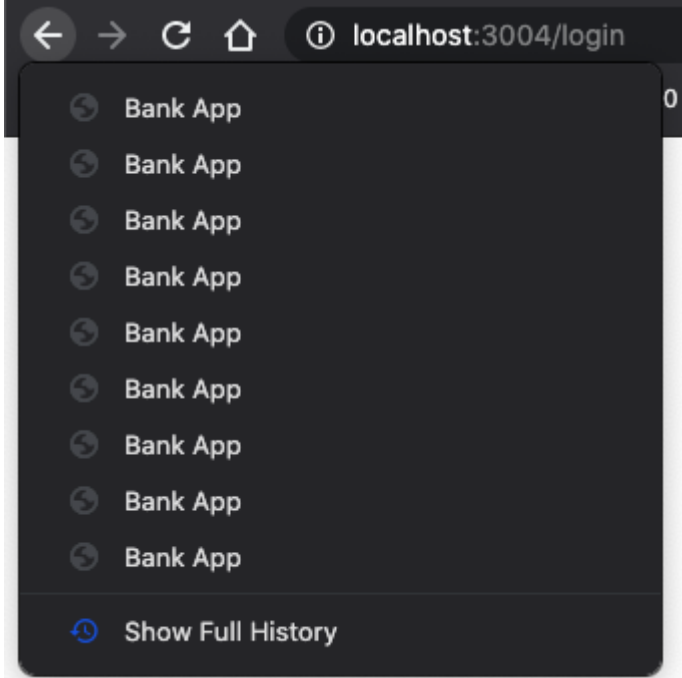
Using the `onclick` attribute bind the `click` event to JavaScript code, here the call to the `navigate()` function.

Try clicking on these links, you should be now able to navigate between the different screens of your app.

✔ The `history.pushState` method is part of the HTML5 standard and implemented in all modern browsers. If you're building a web app for older browsers, there's a trick you can use in place of this API: using a `hash(#)` before the path you can implement routing that works with regular anchor navigation and does not reload the page, as its purpose was to create internal links within a page.

Handling the browser's back and forward buttons

Using the `history.pushState` creates new entries in the browser's navigation history. You can check that by holding the *back button* of your browser, it should display something like this:



If you try clicking on the back button a few times, you'll see that the current URL changes and the history is updated, but the same template keeps being displayed.

That's because the application does not know that we need to call `updateRoute()` every time the history changes. If you take a look at the [history.pushState documentation](#), you can see that if the state changes - meaning that we moved to a different URL - the `popstate` event is triggered. We'll use that to fix that issue.

Task

To make sure the displayed template is updated when the browser history changes, we'll attach a new function that calls `updateRoute()`. We'll do that at the bottom of our `app.js` file:

```
window.onpopstate = () => updateRoute();  
updateRoute();
```

js

Note: we used an [arrow function](#) here to declare our `popstate` event handler for conciseness, but a regular function would work the same.

Here's a refresher video on arrow functions:



 Click the image above for a video about arrow functions.

Now try to use the back and forward buttons of your browsers, and check that the displayed route is correctly updated this time.

Challenge

Add a new template and route for a third page that shows the credits for this app.

Post-Lecture Quiz

[Post-lecture quiz](#)

Review & Self Study

Routing is one of the surprisingly tricky parts of web development, especially as the web moves from page refresh behaviors to Single Page Application page refreshes. Read a little about [how the Azure](#)

[Static Web App service](#) handles routing. Can you explain why some of the decisions described on that document are necessary?

Assignment

[Improve the routing](#)

Build a Banking App Part 2: Build a Login and Registration Form

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

In almost all modern web apps, you can create an account to have your own private space. As multiple users can access a web app at the same time, you need a mechanism to store each user personal data separately and select which information to display information. We won't cover how to manage [user identity securely](#), as it's an extensive topic on its own, but we'll make sure each user is able to create one (or more) bank account on our app.

In this part we'll use HTML forms to add login and registration to our web app. We'll see how to send the data to a server API programmatically, and ultimately how to define basic validation rules for user inputs.

Prerequisite

You need to have completed the [HTML templates and routing](#) of the web app for this lesson. You also need to install [Node.js](#) and [run the server API](#) locally so you can send data to create accounts.

You can test that the server is running properly by executing this command in a terminal:

```
curl http://localhost:5000/api  
# -> should return "Bank API v1.0.0" as a result
```

Form and controls

The `<form>` element encapsulates a section of an HTML document where the user can input and submit data with interactive controls. There are all sorts of user interface (UI) controls that can be used within a form, the most common one being the `<input>` and the `<button>` elements.

There are a lot of different types of `<input>`, for example to create a field where the user can enter its username you can use:

html

```
<input id="username" name="username" type="text">
```

The `name` attribute will be used as the property name when the form data will be sent over. The `id` attribute is used to associate a `<label>` with the form control.

Take a look at the whole list of [<input> types](#) and [other form controls](#) to get an idea of all the native UI elements you can use when building your UI.

✔ Note that `<input>` is an empty element on which you should *not* add a matching closing tag. You can however use the self-closing `<input/>` notation, but it's not required.

The `<button>` element within a form is a bit special. If you do not specify its `type` attribute, it will automatically submit the form data to the server when pressed. Here are the possible `type` values:

- `submit` : The default within a `<form>`, the button triggers the form submit action.
- `reset` : The button resets all the form controls to their initial values.
- `button` : Do not assign a default behavior when the button is pressed. You can then assign custom actions to it using JavaScript.

Task

Let's start by adding a form to the `login` template. We'll need a `username` field and a `Login` button.

```

<template id="login">
  <h1>Bank App</h1>
  <section>
    <h2>Login</h2>
    <form id="loginForm">
      <label for="username">Username</label>
      <input id="username" name="user" type="text">
      <button>Login</button>
    </form>
  </section>
</template>

```

If you take a closer look, you can notice that we also added a `<label>` element here. `<label>` elements are used to add a name to UI controls, such as our username field. Labels are important for the readability of your forms, but also comes with additional benefits:

- By associating a label to a form control, it helps users using assistive technologies (like a screen reader) to understand what data they're expected to provide.
- You can click on the label to directly put focus on the associated input, making it easier to reach on touch-screen based devices.

Accessibility on the web is a very important topic that's often overlooked. Thanks to semantic HTML elements it's not difficult to create accessible content if you use them properly. You can read more about accessibility to avoid common mistakes and become a responsible developer.

Now we'll add a second form for the registration, just below the previous one:

```

<hr/>
<h2>Register</h2>
<form id="registerForm">
  <label for="user">Username</label>
  <input id="user" name="user" type="text">
  <label for="currency">Currency</label>
  <input id="currency" name="currency" type="text" value="$">
  <label for="description">Description</label>
  <input id="description" name="description" type="text">
  <label for="balance">Current balance</label>
  <input id="balance" name="balance" type="number" value="0">

```

```
<button>Register</button>
</form>
```

Using the `value` attribute we can define a default value for a given input. Notice also that the input for `balance` has the `number` type. Does it look different than the other inputs? Try interacting with it.

✅ Can you navigate and interact with the forms using only a keyboard? How would you do that?

Submitting data to the server

Now that we have a functional UI, the next step is to send the data over to our server. Let's make a quick test using our current code: what happens if you click on the *Login* or *Register* button?

Did you notice the change in your browser's URL section?



The default action for a `<form>` is to submit the form to the current server URL using the GET method, appending the form data directly to the URL. This method has some shortcomings though:

- The data sent is very limited in size (about 2000 characters)
- The data is directly visible in the URL (not great for passwords)
- It does not work with file uploads

That's why you can change it to use the POST method which sends the form data to the server in the body of the HTTP request, without any of the previous limitations.

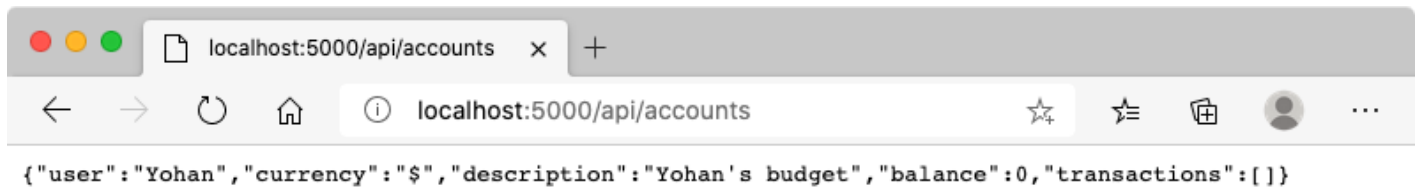
While POST is the most commonly used method to send data over, in some specific scenarios it is preferable to use the GET method, when implementing a search field for example.

Task

Add `action` and `method` properties to the registration form:

```
html
<form id="registerForm" action="//localhost:5000/api/accounts" method="POST"
```


Now try to register a new account with your name. After clicking on the *Register* button you should see something like this:



The screenshot shows a browser window with the address bar containing `localhost:5000/api/accounts`. The page content displays a JSON object: `{"user": "Yohan", "currency": "$", "description": "Yohan's budget", "balance": 0, "transactions": []}`.

If everything goes well, the server should answer your request with a JSON response containing the account data that was created.

✔ Try registering again with the same name. What happens?

Submitting data without reloading the page

As you probably noticed, there's a slight issue with the approach we just used: when submitting the form, we get out of our app and the browser redirects to the server URL. We're trying to avoid all page reloads with our web app, as we're making a Single-page application (SPA).

To send the form data to the server without forcing a page reload, we have to use JavaScript code. Instead of putting an URL in the `action` property of a `<form>` element, you can use any JavaScript code prepended by the `javascript:` string to perform a custom action. Using this also means that you'll have to implement some tasks that were previously done automatically by the browser:

- Retrieve the form data
- Convert and encode the form data to a suitable format
- Create the HTTP request and send it to the server

Task

Replace the registration form `action` with:

```
<form id="registerForm" action="javascript:register()">
```

Open `app.js` add a new function named `register` :

js

```
function register() {  
  const registerForm = document.getElementById('registerForm');  
  const formData = new FormData(registerForm);  
  const data = Object.fromEntries(formData);  
  const jsonData = JSON.stringify(data);  
}
```

Here we retrieve the form element using `getElementById()` and use the `FormData` helper to extract the values from form controls as a set of key/value pairs. Then we convert the data to a regular object using `Object.fromEntries()` and finally serialize the data to `JSON`, a format commonly used for exchanging data on the web.

The data is now ready to be sent to the server. Create a new function named `createAccount` :

js

```
async function createAccount(account) {  
  try {  
    const response = await fetch('://localhost:5000/api/accounts', {  
      method: 'POST',  
      headers: { 'Content-Type': 'application/json' },  
      body: account  
    });  
    return await response.json();  
  } catch (error) {  
    return { error: error.message || 'Unknown error' };  
  }  
}
```

What's this function doing? First, notice the `async` keyword here. This means that the function contains code that will execute **asynchronously**. When used along the `await` keyword, it allows waiting for asynchronous code to execute - like waiting for the server response here - before continuing.

Here's a quick video about `async/await` usage:



 Click the image above for a video about async/await.

We use the `fetch()` API to send JSON data to the server. This method takes 2 parameters:

- The URL of the server, so we put back `//localhost:5000/api/accounts` here.
- The settings of the request. That's where we set the method to `POST` and provide the `body` for the request. As we're sending JSON data to the server, we also need to set the `Content-Type` header to `application/json` so the server know how to interpret the content.

As the server will respond to the request with JSON, we can use `await response.json()` to parse the JSON content and return the resulting object. Note that this method is asynchronous, so we use the `await` keyword here before returning to make sure any errors during parsing are also caught.

Now add some code to the `register` function to call `createAccount()` :

```
const result = await createAccount(jsonData);
```

js

Because we use the `await` keyword here, we need to add the `async` keyword before the `register` function:

```
async function register() {
```

js

Finally, let's add some logs to check the result. The final function should look like this:

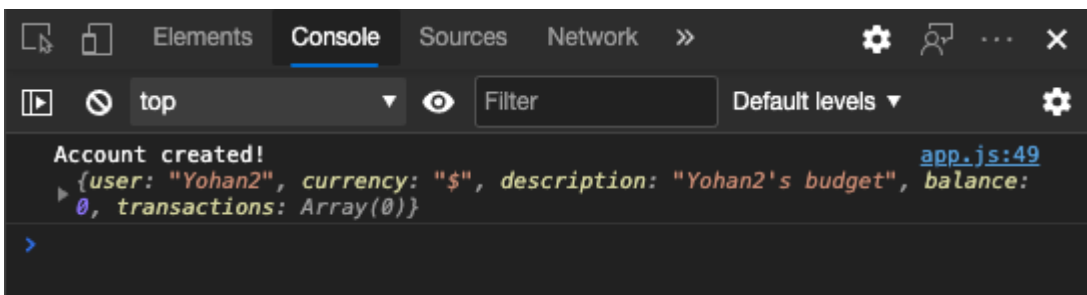
js

```
async function register() {
  const registerForm = document.getElementById('registerForm');
  const formData = new FormData(registerForm);
  const jsonData = JSON.stringify(Object.fromEntries(formData));
  const result = await createAccount(jsonData);

  if (result.error) {
    return console.log('An error occurred:', result.error);
  }

  console.log('Account created!', result);
}
```

That was a bit long but we got there! If you open your [browser developer tools](#), and try registering a new account, you should not see any change on the web page but a message will appear in the console confirming that everything works.



✅ Do you think the data is sent to the server securely? What if someone what was able to intercept the request? You can read about [HTTPS](#) to know more about secure data communication.

Data validation

If you try to register a new account without setting an username first, you can see that the server returns an error with status code [400 \(Bad Request\)](#).

Before sending data to a server it's a good practice to [validate the form data](#) beforehand when possible, to make sure you send a valid request. HTML5 forms controls provides built-in validation using various attributes:

- `required` : the field needs to be filled otherwise the form cannot be submitted.
- `minlength` and `maxlength` : defines the minimum and maximum number of characters in text fields.

- `min` and `max` : defines the minimum and maximum value of a numerical field.
- `type` : defines the kind of data expected, like `number` , `email` , `file` or other built-in types. This attribute may also change the visual rendering of the form control.
- `pattern` : allows to define a regular expression pattern to test if the entered data is valid or not.

Tip: you can customize the look of your form controls depending if they're valid or not using the `:valid` and `:invalid` CSS pseudo-classes.

Task

There are 2 required fields to create a valid new account, the username and currency, the other fields being optional. Update the form's HTML, using both the `required` attribute and text in the field's label to that:

html

```
<label for="user">Username (required)</label>
<input id="user" name="user" type="text" required>
...
<label for="currency">Currency (required)</label>
<input id="currency" name="currency" type="text" value="$" required>
```

While this particular server implementation does not enforce specific limits on the fields maximum length, it's always a good practice to define reasonable limits for any user text entry.

Add a `maxLength` attribute to the text fields:


html

```
<input id="user" name="user" type="text" maxLength="20" required>
...
<input id="currency" name="currency" type="text" value="$" maxLength="5" required>
...
<input id="description" name="description" type="text" maxLength="100">
```

Now if you press the *Register* button and a field does not respect a validation rule we defined, you should see something like this:

Register

Username Currency \$ Description Current balance 0

 Please fill in this field.

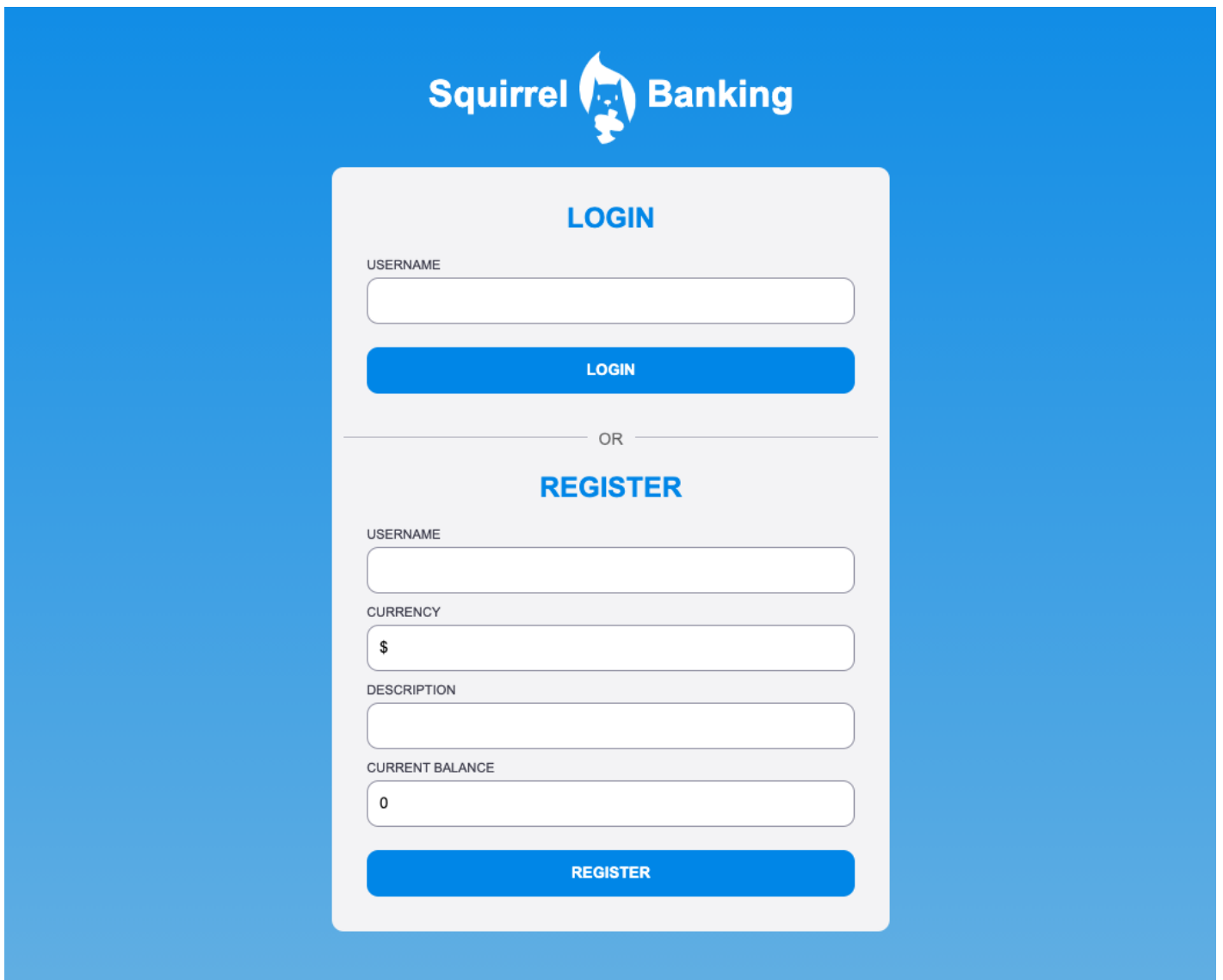
Validation like this performed *before* sending any data to the server is called **client-side** validation. But note that it's not always possible to perform all checks without sending the data. For example, we cannot check here if an account already exists with the same username without sending a request to the server. Additional validation performed on the server is called **server-side** validation.

Usually both need to be implemented, and while using client-side validation improves the user experience by providing instant feedback to the user, server-side validation is crucial to make sure the user data you manipulate is sound and safe.

Challenge

Show an error message in the HTML if the user already exists.

Here's an example of what the final login page can look like after a bit of styling:



The image shows a stylized login and register form for "Squirrel Banking". The form is centered on a blue background. At the top, the "Squirrel Banking" logo is displayed, featuring a squirrel icon. Below the logo, the form is divided into two sections: "LOGIN" and "REGISTER". The "LOGIN" section has a "USERNAME" input field and a "LOGIN" button. The "REGISTER" section has a "USERNAME" input field, a "CURRENCY" input field (with a "\$" symbol), a "DESCRIPTION" input field, and a "CURRENT BALANCE" input field (with a "0" value). A "REGISTER" button is located at the bottom of the form. The form is styled with a light gray background and rounded corners.

[Post-lecture quiz](#)

Review & Self Study

Developers have gotten very creative about their form building efforts, especially regarding validation strategies. Learn about different form flows by looking through [CodePen](#); can you find some interesting and inspiring forms?

Assignment

[Style your bank app](#)

Build a Banking App Part 3: Methods of Fetching and Using Data

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

At the core of every web application there's *data*. Data can take many forms, but its main purpose is always to display information to the user. With web apps becoming increasingly interactive and complex, how the user accesses and interacts with information is now a key part of web development.

In this lesson, we'll see how to fetch data from a server asynchronously, and use this data to display information on a web page without reloading the HTML.

Prerequisite

You need to have built the [Login and Registration Form](#) part of the web app for this lesson. You also need to install [Node.js](#) and [run the server API](#) locally so you get account data.

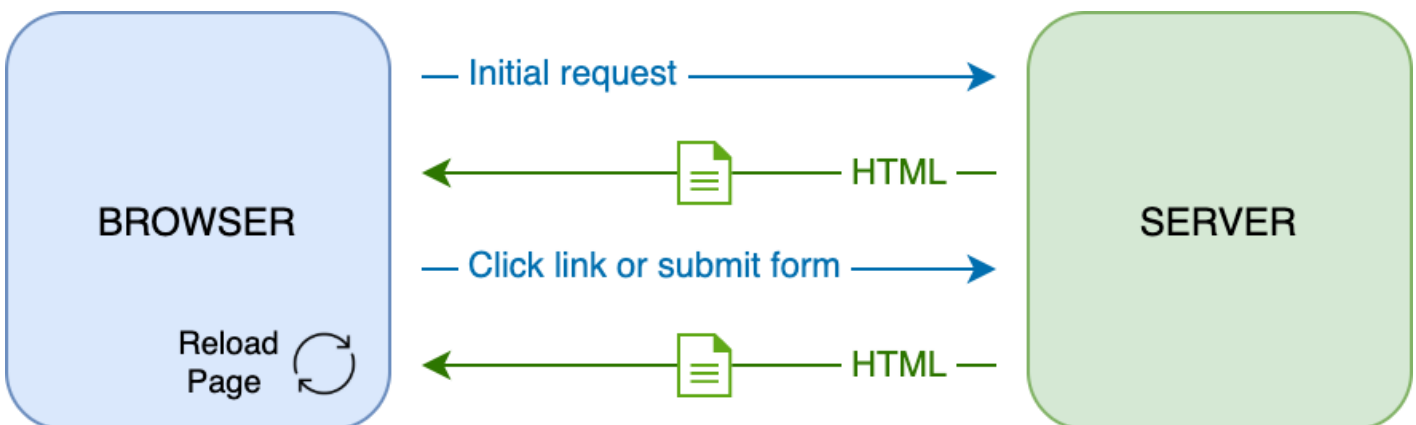
You can test that the server is running properly by executing this command in a terminal:

```
curl http://localhost:5000/api
# -> should return "Bank API v1.0.0" as a result
```

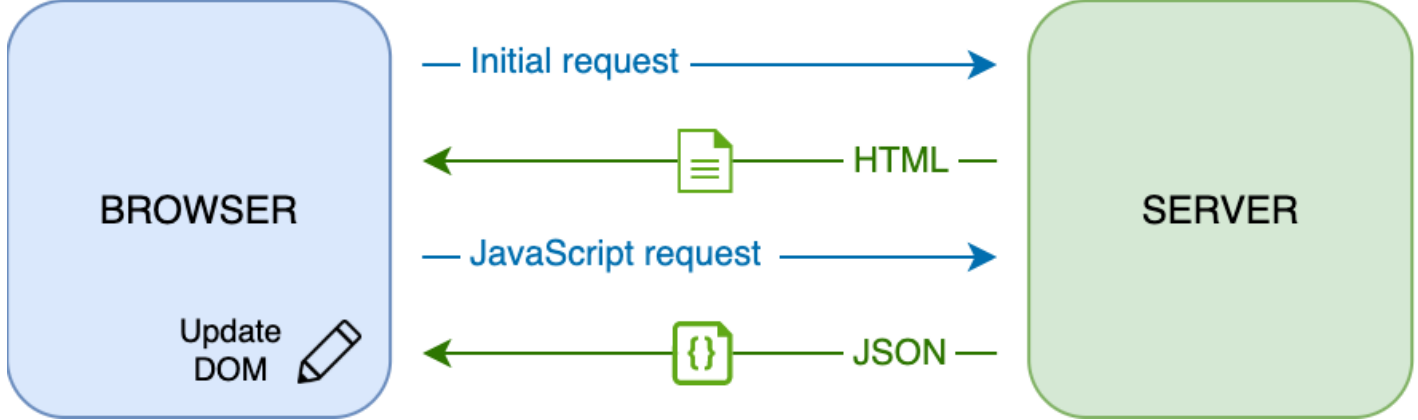
sh

AJAX and data fetching

Traditional web sites update the content displayed when the user selects a link or submits data using a form, by reloading the full HTML page. Every time new data needs to be loaded, the web server returns a brand new HTML page that needs to be processed by the browser, interrupting the current user action and limiting interactions during the reload. This workflow is also called a *Multi-Page Application* or *MPA*.



When web applications started to become more complex and interactive, a new technique called [AJAX \(Asynchronous JavaScript and XML\)](#) emerged. This technique allows web apps to send and retrieve data from a server asynchronously using JavaScript, without having to reload the HTML page, resulting in faster updates and smoother user interactions. When new data is received from the server, the current HTML page can also be updated with JavaScript using the [DOM](#) API. Over time, this approach has evolved into what is now called a *Single-Page Application* or *SPA*.



When AJAX was first introduced, the only API available to fetch data asynchronously was `XMLHttpRequest` . But modern browsers now also implement the more convenient and powerful `Fetch API`, which uses promises and is better suited to manipulate JSON data.

While all modern browsers support the `Fetch API` , if you want your web application to work on legacy or old browsers it's always a good idea to check the [compatibility table on caniuse.com](https://caniuse.com) first.

Task

In [the previous lesson](#) we implemented the registration form to create an account. We'll now add code to login using an existing account, and fetch its data. Open the `app.js` file and add a new `login` function:

```
async function login() {  
  const loginForm = document.getElementById('loginForm')  
  const user = loginForm.user.value;  
}
```

js

Here we start by retrieving the form element with `getElementById()` , and then we get the username from the input with `loginForm.user.value` . Every form control can be accessed by its name (set in the HTML using the `name` attribute) as a property of the form.

In a similar fashion to what we did for the registration, we'll create another function to perform a server request, but this time for retrieving the account data:

```
async function getAccount(user) {  
  try {  
    const response = await fetch('//localhost:5000/api/accounts/' + encode
```

js

```

    return await response.json();
  } catch (error) {
    return { error: error.message || 'Unknown error' };
  }
}

```

We use the `fetch` API to request the data asynchronously from the server, but this time we don't need any extra parameters other than the URL to call, as we're only querying data. By default, `fetch` creates a GET HTTP request, which is what we are seeking here.

✅ `encodeURIComponent()` is a function that escapes special characters for URL. What issues could we possibly have if we do not call this function and use directly the `user` value in the URL?

Let's now update our `login` function to use `getAccount` :

```

async function login() {
  const loginForm = document.getElementById('loginForm')
  const user = loginForm.user.value;
  const data = await getAccount(user);

  if (data.error) {
    return console.log('loginError', data.error);
  }

  account = data;
  navigate('/dashboard');
}

```

js

First, as `getAccount` is an asynchronous function we need to match it with the `await` keyword to wait for the server result. As with any server request, we also have to deal with error cases. For now we'll only add a log message to display the error, and come back to it later.

Then we have to store the data somewhere so we can later use it to display the dashboard informations. Since the `account` variable does not exist yet, we'll create a global variable for it at the top of our file:

```

let account = null;

```

js

After the user data is saved into a variable we can navigate from the *login* page to the *dashboard* using the `navigate()` function we already have.

Finally, we need to call our `login` function when the login form is submitted, by modifying the HTML:

html

```
<form id="loginForm" action="javascript:login()">
```

Test that everything is working correctly by registering a new account and trying to login using the same account.

Before moving on to the next part, we can also complete the `register` function by adding this at the bottom of the function:

js

```
account = result;
navigate('/dashboard');
```

✅ Did you know that by default, you can only call server APIs from the *same domain and port* than the web page you are viewing? This is security mechanism enforced by browsers. But wait, our web app is running on `localhost:3000` whereas the server API is running on `localhost:5000`, why does it work? By using a technique called Cross-Origin Resource Sharing (CORS), it is possible to perform cross-origin HTTP requests if the server adds special headers to the response, allowing exceptions for specific domains.

Learn more about APIs by taking this [lesson](#)

Update HTML to display data

Now that we have the user data, we have to update the existing HTML to display it. We already know how to retrieve an element from the DOM using for example `document.getElementById()`. After you have a base element, here are some APIs you can use to modify it or add child elements to it:

- Using the `textContent` property you can change the text of an element. Note that changing this value removes all the element's children (if there's any) and replaces it with the text provided. As such, it's also an efficient method to remove all children of a given element by assigning an empty string `''` to it.
- Using `document.createElement()` along with the `append()` method you can create and attach one or more new child elements.

✔ Using the `innerHTML` property of an element it's also possible to change its HTML contents, but this one should be avoided as it's vulnerable to cross-site scripting (XSS) attacks.

Task

Before moving on to the dashboard screen, there's one more thing we should do on the *login* page. Currently, if you try to login with a username that does not exist, a message is shown in the console but for a normal user nothing changes and you don't know what's going on.

Let's add a placeholder element in the login form where we can display an error message if needed. A good place would be just before the login `<button>` :

html

```
...  
<div id="loginError"></div>  
<button>Login</button>  
...
```

This `<div>` element is empty, meaning that nothing will be displayed on the screen until we add some content to it. We also give it an `id` so we can retrieve it easily with JavaScript.

Go back to the `app.js` file and create a new helper function `updateElement` :

js

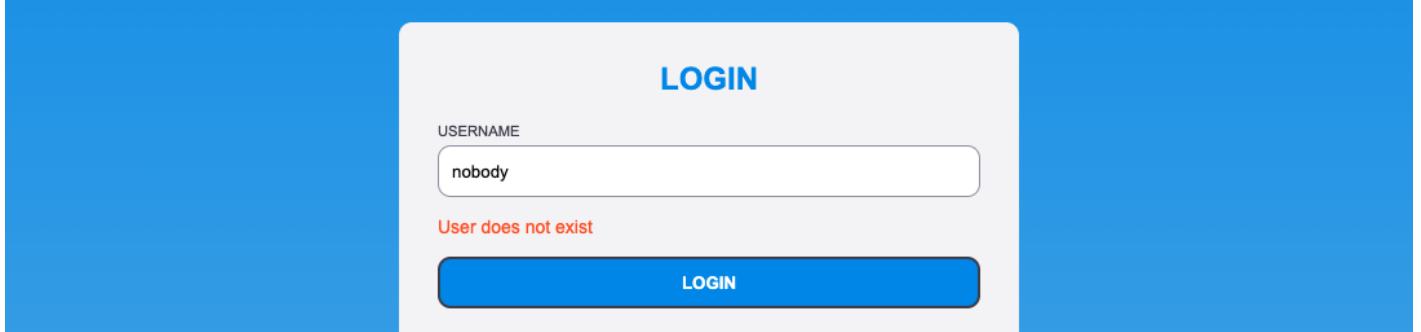
```
function updateElement(id, text) {  
  const element = document.getElementById(id);  
  element.textContent = text;  
}
```

This one is quite straightforward: given an element `id` and `text`, it will update the text content of the DOM element with the matching `id`. Let's use this method in place of the previous error message in the `login` function:

js

```
if (data.error) {  
  return updateElement('loginError', data.error);  
}
```

Now if you try to login with an invalid account, you should see something like this:



Now we have error text that shows up visually, but if you try it with a screen reader you'll notice that nothing is announced. In order for text that is dynamically added to a page to be announced by screen readers, it will need to use something called a [Live Region](#). Here we're going to use a specific type of live region called an alert:

html

```
<div id="loginError" role="alert"></div>
```

Implement the same behavior for the `register` function errors (don't forget to update the HTML).

Display information on the dashboard

Using the same techniques we've just seen, we'll also take care of displaying the account information on the dashboard page.

This is what an account object received from the server looks like:

json

```
{
  "user": "test",
  "currency": "$",
  "description": "Test account",
  "balance": 75,
  "transactions": [
    { "id": "1", "date": "2020-10-01", "object": "Pocket money", "amount": 100 },
    { "id": "2", "date": "2020-10-03", "object": "Book", "amount": -10 },
    { "id": "3", "date": "2020-10-04", "object": "Sandwich", "amount": -5 }
  ],
}
```

Note: to make your life easier, you can use the pre-existing `test` account that's already populated with data.

Task

Let's start by replacing the "Balance" section in the HTML to add placeholder elements:

html

```
<section>
  Balance: <span id="balance"></span><span id="currency"></span>
</section>
```

We'll also add a new section just below to display the account description:

html

```
<h2 id="description"></h2>
```

✅ Since the account description functions as a title for the content underneath it, it is marked up semantically as a heading. Learn more about how [heading structure](#) is important for accessibility, and take a critical look at the page to determine what else could be a heading.

Next, we'll create a new function in `app.js` to fill in the placeholder:

js

```
function updateDashboard() {
  if (!account) {
    return navigate('/login');
  }

  updateElement('description', account.description);
  updateElement('balance', account.balance.toFixed(2));
  updateElement('currency', account.currency);
}
```

First, we check that we have the account data we need before going further. Then we use the `updateElement()` function we created earlier to update the HTML.

To make the balance display prettier, we use the method `toFixed(2)` to force displaying the value with 2 digits after the decimal point.

Now we need to call our `updateDashboard()` function everytime the dashboard is loaded. If you already finished the [lesson 1 assignment](#) this should be straightforward, otherwise you can use the following implementation.

Add this code to the end of the `updateRoute()` function:

```
if (typeof route.init === 'function') {
  route.init();
}
```

js

And update the routes definition with:

```
const routes = {
  '/login': { templateId: 'login' },
  '/dashboard': { templateId: 'dashboard', init: updateDashboard }
};
```

js

With this change, every time the dashboard page is displayed, the function `updateDashboard()` is called. After a login, you should then be able to see the account balance, currency and description.

Create table rows dynamically with HTML templates

In the [first lesson](#) we used HTML templates along with the `appendChild()` method to implement the navigation in our app. Templates can also be smaller and used to dynamically populate repetitive parts of a page.

We'll use a similar approach to display the list of transactions in the HTML table.

Task

Add a new template in the HTML `<body>` :

```
<template id="transaction">
  <tr>
    <td></td>
    <td></td>
    <td></td>
```

html

```
</tr>  
</template>
```

This template represents a single table row, with the 3 columns we want to populate: *date*, *object* and *amount* of a transaction.

Then, add this `id` property to the `<tbody>` element of the table within the dashboard template to make it easier to find using JavaScript:

```
<tbody id="transactions"></tbody>
```

html

Our HTML is ready, let's switch to JavaScript code and create a new function `createTransactionRow` :

```
function createTransactionRow(transaction) {  
  const template = document.getElementById('transaction');  
  const transactionRow = template.content.cloneNode(true);  
  const tr = transactionRow.querySelector('tr');  
  tr.children[0].textContent = transaction.date;  
  tr.children[1].textContent = transaction.object;  
  tr.children[2].textContent = transaction.amount.toFixed(2);  
  return transactionRow;  
}
```

js

This function does exactly what its names implies: using the template we created earlier, it creates a new table row and fills in its contents using transaction data. We'll use this in our `updateDashboard()` function to populate the table:

```
const transactionsRows = document.createDocumentFragment();  
for (const transaction of account.transactions) {  
  const transactionRow = createTransactionRow(transaction);  
  transactionsRows.appendChild(transactionRow);  
}  
updateElement('transactions', transactionsRows);
```

js

Here we use the method `document.createDocumentFragment()` that creates a new DOM fragment on which we can work, before finally attaching it to our HTML table.

There's still one more thing we have to do before this code can work, as our `updateElement()` function currently supports text content only. Let's change its code a bit:


```
function updateElement(id, textOrNode) {  
  const element = document.getElementById(id);  
  element.textContent = ''; // Removes all children  
  element.append(textOrNode);  
}
```

We use the `append()` method as it allows to attach either text or [DOM Nodes](#) to a parent element, which is perfect for all our use cases.

If you try using the `test` account to login, you should now see a transaction list on the dashboard



Challenge

Work together to make the dashboard page look like a real banking app. If you already styled your app, try to use [media queries](#) to create a [responsive design](#) working nicely on both desktop and mobile devices.

Here's an example of a styled dashboard page:



BALANCE

231.00\$

Yohan's budget

[ADD TRANSACTION](#)

Date	Object	Amount
2020-09-10	Allowance	100.00
2020-09-12	JS book	-20.00
2020-09-14	Food	-25.00
2020-09-15	Grandma's gift	50.00
2020-09-16	Cinema	-15.00
2020-09-19	Video game	-59.00

Post-Lecture Quiz

[Post-lecture quiz](#)

Assignment

[Refactor and comment your code](#)

Build a Banking App Part 4: Concepts of State Management

Pre-Lecture Quiz

[Pre-lecture quiz](#)

Introduction

As a web application grows, it becomes a challenge to keep track of all data flows. Which code gets the data, what page consumes it, where and when does it need to be updated...it's easy to end up with messy code that's difficult to maintain. This is especially true when you need to share data among different pages of your app, for example user data. The concept of *state management* has always existed in all kinds of programs, but as web apps keep growing in complexity it's now a key point to think about during development.

In this final part, we'll look over the app we built to rethink how the state is managed, allowing support for browser refresh at any point, and persisting data across user sessions.

Prerequisite

You need to have completed the [data fetching](#) part of the web app for this lesson. You also need to install [Node.js](#) and [run the server API](#) locally so you can manage account data.

You can test that the server is running properly by executing this command in a terminal:

```
curl http://localhost:5000/api
# -> should return "Bank API v1.0.0" as a result
```

sh

Rethink state management

In the [previous lesson](#), we introduced a basic concept of state in our app with the global `account` variable which contains the bank data for the currently logged in user. However, our current implementation has some flaws. Try refreshing the page when you're on the dashboard. What happens?

There's 3 issues with the current code:

- The state is not persisted, as a browser refresh takes you back to the login page.
- There are multiple functions that modify the state. As the app grows, it can make it difficult to track the changes and it's easy to forget updating one.
- The state is not cleaned up, so when you click on *Logout* the account data is still there even though you're on the login page.

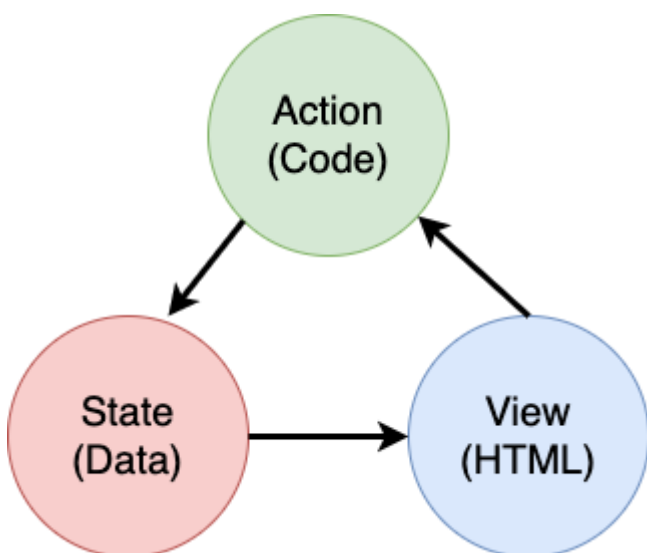
We could update our code to tackle these issues one by one, but it would create more code duplication and make the app more complex and difficult to maintain. Or we could pause for a few minutes and rethink our strategy.

What problems are we really trying to solve here?

State management is all about finding a good approach to solve these two particular problems:

- How to keep the data flows in an app understandable?
- How to keep the state data always in sync with the user interface (and vice versa)?

Once you've taken care of these, any other issues you might have may either be fixed already or have become easier to fix. There are many possible approaches for solving these problems, but we'll go with a common solution that consists of **centralizing the data and the ways to change it**. The data flows would go like this:



We won't cover here the part where the data automatically triggers the view update, as it's tied to more advanced concepts of Reactive Programming. It's a good follow-up subject if you're up to a deep dive.

✅ There are a lot of libraries out there with different approaches to state management, Redux being a popular option. Take a look at the concepts and patterns used as it's often a good way to learn what potential issues you may be facing in large web apps and how it can be solved.

Task

We'll start with a bit of refactoring. Replace the `account` declaration:

```
let account = null;
```

js

With:

```
let state = {  
  account: null  
};
```

js

The idea is to *centralize* all our app data in a single state object. We only have `account` for now in the state so it doesn't change much, but it creates a path for evolutions.

We also have to update the functions using it. In the `register()` and `login()` functions, replace `account = ...` with `state.account = ...`;

At the top of the `updateDashboard()` function, add this line:

```
const account = state.account;
```

js

This refactoring by itself did not bring much improvements, but the idea was to lay out the foundation for the next changes.

Track data changes

Now that we have put in place the `state` object to store our data, the next step is centralize the updates. The goal is to make it easier to keep track of any changes and when they happen.

To avoid having changes made to the `state` object, it's also a good practice to consider it *immutable*, meaning that it cannot be modified at all. It also means that you have to create a new state object if you want to change anything in it. By doing this, you build a protection about potentially unwanted side effects, and open up possibilities for new features in your app like implementing undo/redo, while also making it easier to debug. For example, you could log every change made to the state and keep a history of the changes to understand the source of a bug.

In JavaScript, you can use `Object.freeze()` to create an immutable version of an object. If you try to make changes to an immutable object, an exception will be raised.

✔ Do you know the difference between a *shallow* and a *deep* immutable object? You can read about it [here](#).

Task

Let's create a new `updateState()` function:

```
function updateState(property, newData) {
  state = Object.freeze({
    ...state,
    [property]: newData
  });
}
```

js

In this function, we're creating a new state object and copy data from the previous state using the `spread(...)_operator`. Then we override a particular property of the state object with the new data using the `bracket notation` `[property]` for assignment. Finally, we lock the object to prevent modifications using `Object.freeze()`. We only have the `account` property stored in the state for now, but with this approach you can add as many properties as you need in the state.

We'll also update the `state` initialization to make sure the initial state is frozen too:

```
let state = Object.freeze({
  account: null
});
```

js

After that, update the `register` function by replacing the `state.account = result;` assignment with:

```
updateState('account', result);
```

js

Do the same with the `login` function, replacing `state.account = data;` with:

```
updateState('account', data);
```

js

We'll now take the chance to fix the issue of account data not being cleared when the user clicks on `Logout`.

Create a new function `logout()` :

js

```
function logout() {  
  updateState('account', null);  
  navigate('/login');  
}
```

In `updateDashboard()`, replace the redirection `return navigate('/login');` with `return logout();`

Try registering a new account, logging out and in again to check that everything still works correctly.

Tip: you can take a look at all state changes by adding `console.log(state)` at the bottom of `updateState()` and opening up the console in your browser's development tools.

Persist the state

Most web apps need to persist data to be able to work correctly. All the critical data is usually stored on a database and accessed via a server API, like as the user account data in our case. But sometimes, it's also interesting to persist some data on the client app that's running in your browser, for a better user experience or to improve loading performance.

When you want to persist data in your browser, there are a few important questions you should ask yourself:

- *Is the data sensitive?* You should avoid storing any sensitive data on client, such as user passwords.
- *For how long do you need to keep this data?* Do you plan to access this data only for the current session or do you want it to be stored forever?

There are multiple ways of storing information inside a web app, depending on what you want to achieve. For example, you can use the URLs to store a search query, and make it shareable between users. You can also use [HTTP cookies](#) if the data needs to be shared with the server, like [authentication](#) information.

Another option is to use one of the many browser APIs for storing data. Two of them are particularly interesting:

- `localStorage` : a Key/Value store allowing to persist data specific to the current web site across different sessions. The data saved in it never expires.
- `sessionStorage` : this one works the same as `localStorage` except that the data stored in it is cleared when the session ends (when the browser is closed).

Note that both these APIs only allow to store strings. If you want to store complex objects, you will need to serialize it to the JSON format using `JSON.stringify()` .

✅ If you want to create a web app that does not work with a server, it's also possible to create a database on the client using the IndexedDB API. This one is reserved for advanced use cases or if you need to store significant amount of data, as it's more complex to use.

Task

We want our users stay logged in until they explicitly click on the *Logout* button, so we'll use `localStorage` to store the account data. First, let's define a key that we'll use to store our data.

```
const storageKey = 'savedAccount';
```

js

Then add this line at the end of the `updateState()` function:

```
localStorage.setItem(storageKey, JSON.stringify(state.account));
```

js

With this, the user account data will be persisted and always up-to-date as we centralized previously all our state updates. This is where we begin to benefit from all our previous refactors 😊 .

As the data is saved, we also have to take care of restoring it when the app is loaded. Since we'll begin to have more initialization code it may be a good idea to create a new `init` function, that also includes our previous code at the bottom of `app.js` :

```
function init() {  
  const savedAccount = localStorage.getItem(storageKey);  
  if (savedAccount) {  
    updateState('account', JSON.parse(savedAccount));  
  }  
  
  // Our previous initialization code  
  window.onpopstate = () => updateRoute();  
  updateRoute();  
}
```

js


```
init();
```

Here we retrieve the saved data, and if there's any we update the state accordingly. It's important to do this *before* updating the route, as there might be code relying on the state during the page update.

We can also make the *Dashboard* page our application default page, as we are now persisting the account data. If no data is found, the dashboard takes care of redirecting to the *Login* page anyways. In `updateRoute()`, replace the fallback `return navigate('/login');` with `return navigate('/dashboard');`.

Now login in the app and try refreshing the page. You should stay on the dashboard. With that update we've taken care of all our initial issues...

Refresh the data

...But we might also have a created a new one. Oops!

Go to the dashboard using the `test` account, then run this command on a terminal to create a new transaction:

```
sh
curl --request POST \
  --header "Content-Type: application/json" \
  --data "{ \"date\": \"2020-07-24\", \"object\": \"Bought book\", \"amount\": 100 }" \
  http://localhost:5000/api/accounts/test/transactions
```

Try refreshing your the dashboard page in the browser now. What happens? Do you see the new transaction?

The state is persisted indefinitely thanks to the `localStorage`, but that also means it's never updated until you log out of the app and log in again!

One possible strategy to fix that is to reload the account data every time the dashboard is loaded, to avoid stall data.

Task

Create a new function `updateAccountData` :

```

async function updateAccountData() {
  const account = state.account;
  if (!account) {
    return logout();
  }

  const data = await getAccount(account.user);
  if (data.error) {
    return logout();
  }

  updateState('account', data);
}

```

This method checks that we are currently logged in then reloads the account data from the server.

Create another function named `refresh` :

```

async function refresh() {
  await updateAccountData();
  updateDashboard();
}

```

This one updates the account data, then takes care of updating the HTML of the dashboard page. It's what we need to call when the dashboard route is loaded. Update the route definition with:

```

const routes = {
  '/login': { templateId: 'login' },
  '/dashboard': { templateId: 'dashboard', init: refresh }
};

```

Try reloading the dashboard now, it should display the updated account data.

Challenge

Now that we reload the account data every time the dashboard is loaded, do you think we still need to persist *all the account data*?

Try working together to change what is saved and loaded from `localStorage` to only include what is absolutely required for the app to work.

Post-Lecture Quiz

[Post-lecture quiz](#)

Assignment

[Implement "Add transaction" dialog](#)

Here's an example result after completing the assignment:

