23.1. JavaScript and the Browser

23.1.1. Taking JavaScript on the Web

So far, we have created web pages with HTML and CSS. These pages have been **static**, meaning that the page appears the same each time it loads. However, you may find that you want to create a web page that changes after it's been loaded. In order to create such a page, you would use JavaScript. Web pages that can change after loading in the browser are called **dynamic**. This is useful to programmers and users alike because they can interact with an application without refreshing the page. Having to constantly refresh the page would be a poor experience for the user and JavaScript helps programmers alleviate this burden.

Example

When you are on a social media page, you may like someone's post. When you do like their post, you may notice that several things happen. The counter of how many likes the post has received increases by one and the like button may change color to indicate to you that you liked the post. This is an example of how JavaScript could be used to create an application that dynamically updates without the page having to be refreshed.

We have been running all of our JavaScript code in Node.js, but now it is time to use JavaScript in the browser to make dynamic web pages. Node.js, or just Node, is a JavaScript interpreter with access to lots of different JavaScript libraries. Each browser has their own engine for running JavaScript. JavaScript run in the browser is called client-side JavaScript. Firefox uses an engine called Spider Monkey to run client-side JavaScript.

23.1.2. The `<script>` Tag

In the HTML chapter, we learned that an HTML page is made up of elements that are written as tags. Those elements have different purposes. The `script` element's purpose is to include JavaScript into the web page. A `<script>` tag can contain JavaScript code inside of it or reference an external JavaScript file.

23.1.2.1. JavaScript Console

Using the Developer Tools, you can access a JavaScript console. There, you can mess around with fun JavaScript statements or you can use it to see the outputs of the client-side JavaScript you have written.

23.1.2.2. Inline JavaScript

Example

Notice the `<script>` tag below contains JavaScript code that will be executed by the browser.

```
<!DOCTYPE html>
<html>
<head>
    <title>Embedded JavaScript Example</title>
    <script>
        // JavaScript code goes here!
        console.log("Hello from inside the web page!");
    </script>
</head>
<body>
    contents
</body>
```

```
13</html>
```

Console Output

```
Hello from inside the web page!
```

23.1.2.3. External JavaScript

Some programmers have large amounts of JavaScript to add to an HTML document. Using an external JavaScript file can help in these cases. You can still use the `<script>` tag to include the JavaScript file within the HTML document. In this case, you'll need to use the `src` attribute for the path to the JavaScript file.

Example

This is how the HTML file for the web page might look if we wanted to link an external JavaScript file.

```
<!DOCTYPE html>
   <html>
   <head>
      <title>External JavaScript Example</title>
      <script src = "myjs.js"></script>
   </head>
   <body>
      <!-- content -->
   </body>
   </html>
```

Then the JavaScript file, `myjs.js` might look something like this.

```
// JavaScript code goes here!
console.log("Hello from inside the web page");
```

Note

You can use the `<script>` tag to reference JavaScript files hosted on external servers. Some of these JavaScript files will be files that you have not written yourself but you will want to include in your application.

23.1.3. Check Your Understanding

Question

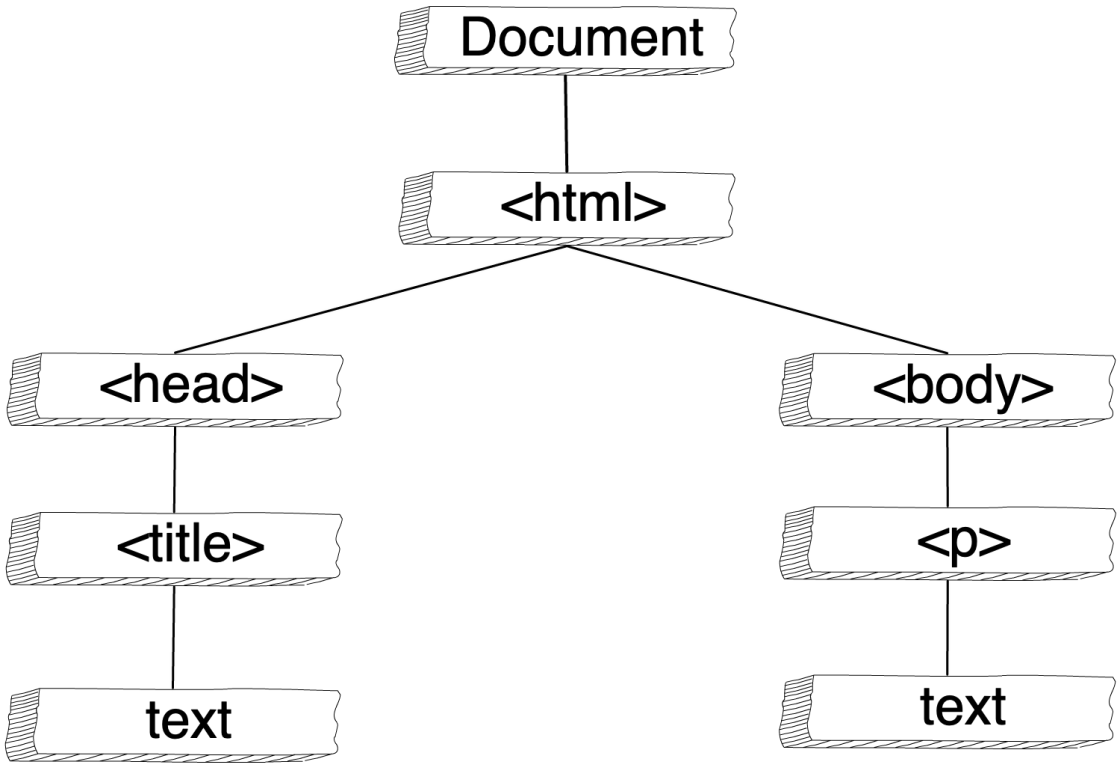
What is the difference between dynamic and static web pages?

Question

Does Node.js run in the browser environment?

23.2. The DOM

You may remember from earlier chapters that classes represent specific entities. The **Document Object Model (DOM)** is a set of objects that represent the browser and the documents that make up the web page. The DOM objects are instances of classes that model the structure of the browser, HTML document, HTML elements, element attributes, and CSS. The below figure depicts the parent-child relationships between the DOM objects that make up a web page.



23.2.1. Global DOM Variables

To utilize the DOM in JavaScript, we need to use the DOM global variables. In the next section, we will learn more about the DOM global variables, including their type. For now, let's get used to the idea of using JavaScript to interact with the DOM.

To start, we are going to use the `window` and `document` global variables. As mentioned above, we will go into more detail on these variables and what they are later.

Example

Here, the `window` and `document` variables are used to print information about the web page to the browser's console.

```
<!DOCTYPE html>
<html>
<head>
    <title>Using DOM Variables</title>
    <script>
```

```
        console.log("the page title:", document.title);
        console.log("the protocol:", window.location.protocol);
    </script>
</head>
<body>
    contents
</body>
</html>
```

Console Output

```
the page title: Using DOM Variables
the protocol: https:
```

23.2.2. Dynamic Web Page Using the DOM

The DOM plays a key part in making web pages dynamic. Since the DOM is a JavaScript representation of the web page, you can use JavaScript to alter the DOM and consequently, the web page. The browser will re-render the web page anytime changes are made via the DOM.

Note

Rendering is not the same action as loading.
In order to add or edit HTML elements with code, we need to be able to access them. The method `document.getElementById` will search for a matching element and return a reference to it. We will go into more detail on how this method works in the next section.

Example

We can use `document.getElementById` and `element.append` to add text to a `<p>` tag.

```
<!DOCTYPE html>
<html>
<head>
    <title>Add content using DOM</title>
</head>
<body>
    <p id="main-text">Words about things...</p>
    <script>
        let p = document.getElementById("main-text");
        p.append("More words about things");
        console.log(p.innerHTML);
    </script>
</body>
</html>
```

Console Output

```
Words about things... More words about things
```

23.2.3. Where to Put the `<script>`

In the previous example, notice the `<script>` tag is placed below the `<p>` tag in the HTML document. HTML documents are executed top down. Therefore, a `<script>` tag must come after any other elements that will be affected by the code inside the `<script>`. Later in the chapter, we will learn about another way to handle this.

23.2.4. Check Your Understanding

Question

What do the DOM objects represent?

a. Word documents you have downloaded
b. Directives of memory
c. The browser window, HTML document, and the elements

Question

What is the value of `p.innerHTML`?

```
 <p id="demo-text">Hello friend</p>
<script>
  let p = document.getElementById("demo-text");
  console.log(p.innerHTML);
</script>

```

23.3. More DOM Methods and Properties

The following sections are a summary of some DOM classes, methods, and properties. A more complete list can be found in the reference links below. You do NOT need to memorize everything on these reference pages. We are providing them to you as a guide for your future studies of the DOM.

1. W3Schools DOM reference
2. MDN DOM reference

23.3.1. Window

The global variable `window` is an instance of the `Window` class. The `Window` class represents the browser window. In the case of multi-tabbed browsers, the global `window` variable represents the specific tab in which the JavaScript code is running.

Window Properties and Methods

Method or Property	Syntax	Description
alert	`window.alert("String message")`	Displays a dialog box with a message and an "ok" button to close the box.
confirm	`window.confirm("String message")`	Displays a dialog box with a message and returns `true` if user clicks "ok" and `false` if user clicks "cancel".
location	`window.location`	Object that represents and alters the web address of the window or tab.
console	`window.console`	Represents the debugging console. Most common and basic use is `window.console.log()`.

Note

When using JavaScript in the browser environment, methods and properties defined on the `Window` class are exposed as global functions and variables. An example of this is `window.console.log()` is accessible by referencing `console.log()` directly.

23.3.2. Document

The global `document` variable is an instance of the `Document` class. The `Document` class represents the HTML web page that is read and displayed by the browser. The `Document` class provides properties and methods to find, add, remove, and alter HTML elements inside on the web page.

Document Properties and Methods

Method or Property	Syntax	Description
title	`document.title`	Read or set the title of the document.
getElementById	`document.getElementById("example-id")`	Returns a reference to the element that's `id` attribute matches the given string value.
querySelector	`document.querySelector("css selector")`	Returns the first element that matches the given CSS selector.
querySelectorAll	`document.querySelectorAll("css selector")`	Returns a list of elements that match the given CSS selector.

Note

`querySelector` and `querySelectorAll` use the CSS selector pattern to find matching elements. The pattern passed in must be a valid CSS selector. Elements will be found and returned the same way that elements are selected to have CSS rules applied.

23.3.3. Element

HTML documents are made up of a tree of elements. The `Element` class represents an HTML element.

Element Properties and Methods

Method or Property	Syntax	Description
getAttribute	`element.getAttribute("id")`	Returns the value of the attribute.
setAttribute	`element.setAttribute("id", "string-value")`	Sets the attribute to the given value.

Element Properties and Methods

Method or Property	Syntax	Description
style	`element.style.color`	Object that allows reading and setting *INLINE* CSS properties.
innerHTML	`element.innerHTML`	Reads or sets the HTML inside an element.

23.3.4. Check Your Understanding

Question

What value will `response` have if the user clicks *Cancel*?

```
let response = window.confirm("String message")
```

Question

Which of these are TRUE about selecting DOM elements?

a. You can select elements by *CSS class* name
b. You can select elements by *id attribute* value
c. You can select elements by *tag* name
d. All of the above

Question

What is the difference between the document and window variables?

23.4. Events

Have you ever thought about how programs respond to interactions from users and other programs? **Events** are code representations of these interactions that need to be responded to.

In programming, events are triggered and then handled.

Events in programming are triggered and handled. **Triggering** an event is the act of causing an event to be sent. **Handling** an event is receiving the event and performing an action in response.

23.4.1. JavaScript and Events

JavaScript is an event-driven programming language. **Event-driven** is a programming pattern where the flow of the program is determined by a series of events. JavaScript uses events to handle user interaction and make web pages dynamic. JavaScript also uses events to know when the state of the web page components change.

23.4.2. DOM Events

Running JavaScript in the browser requires a specific set of events that relate to loading, styling, and displaying HTML elements. Objects in the DOM have event handling built right into them.

Some elements, such as `a`, have default functionality that handles certain events. An example of default event handling is when a user clicks on an `<a>` tag, the browser will navigate to the address in the `href` attribute.

Note

The DOM defines *numerous* events. Each element type does NOT support every event type. The kinds of events that each element supports relate to how the element is used.

23.4.3. Handling Events

Feature-rich web applications rely on more than the default event handling provided by the DOM. We can add custom interactivity with the users by attaching event handlers to HTML elements and then writing the event handler code.

To write a handler, you need to tell the browser what to do when a certain event happens. DOM elements use the *on event* naming convention when declaring event handlers.

The first way we will handle events is to declare the event handler in HTML, this is often referred to as an **inline event handler**. For example, when defining what happens when a `button` element is clicked, the `onclick` attribute is used. This naming convention can be read as: *On click of the button, print a message to the console.*

Example

```
1<!DOCTYPE html>
2<html>
3<head>
4      <title>Button click handler</title>
5</head>
6<body>
7      <button onclick="console.log('you rang...');">Ring Bell</button>
8</body>
9</html>
```

Console Output (if button is clicked)

```
you rang...
```

Tip

Notice the use of single quotes around `'you rang...'`. When declaring the value of an attribute to be a string, you must use single quotes `'` inside the double quotes `"`.

Note

`button` elements represent a clickable entity. `button` elements have default *click* handling behavior related to `form` elements. That we will get into in a later chapter. For now, we will be defining the *click* handler behavior.

Any JavaScript function can be used as the event handler. That means any defined functions can be used. Because programmers can write functions to do whatever their hearts desire, defined functions as event handlers allow for more functionality to occur when an event is handled.

Example

A function `youRang()` is defined and used as the event handler for when the button is clicked.

```
<!DOCTYPE html>
<html>
<head>
      <title>Button click handler</title>
   <script>
         function youRang() {
            document.getElementById("main-text").innerHTML += "you rang...";
            console.log("you rang...");
         }
   </script>
</head>
<body>
   <h1>demo header</h1>
   <p id="main-text" class="orange" style="font-weight: bold;">
         a bunch of really valuable text...
   </p>
   <button onclick="youRang();">Ring Bell</button>
</body>
</html>
```

Result (if button is clicked)

```
effect on page: adds "you rang..." to <p>
output in console: you rang...
```

Warning

When defining handlers via HTML, be very careful to type the function name correctly. If the function name is incorrect, the event will not be handled. No warning is given, the event is silently ignored.

23.4.4. Check Your Understanding

Question

What does an *event* represent in the browser JavaScript environment?

Question

Why is JavaScript considered an *event-driven* language?

Question

Receiving an event and responding to it is known as?

a. Holding an event
b. Having an event
c. Handling an event

23.5. Event Listeners

Using inline event handling is a good way to get started handling events. A second way to handle events uses the DOM objects and methods. Remember, the DOM is an object representation of the entire web page. The DOM allows us to use JavaScript to configure our event handlers. The event handling declaration will no longer be in the HTML element attribute, but will instead be inside `<script>` tags or in an external JavaScript file.

23.5.1. Add Event Handlers in JavaScript

Before we add event handlers in JavaScript, we need to learn a new vocabulary term related to events in programming. A **listener** is another name for an event handler. The term listener refers to the code *listening* for the event to occur. If the code *hears* the event, then the event is handled.

`addEventListener` is used to add an event handler, aka *listener*. `addEventListener` is a method available on instances of `Window`, `Document`, and `Element` classes.

```
anElement.addEventListener("eventName", aFunction);
```

`anElement` is a reference to a DOM element object. `"eventName"` is the name of an event that the variable `anElement` supports. `aFunction` is a reference to a function. To start, we are going to use a *named function*.

Example

We want to set the named function `youRang` as the *click* handler for the `button` element. Notice that the value passed in as the event name is `"click"` instead of `"onclick"`.

```
<!DOCTYPE html>
<html>
<head>
   <title>Use addEventListener</title>
</head>
<body>
   <p id="main-text" class="orange" style="font-weight: bold;">
      a bunch of really valuable text...
   </p>
   <button id="ring-button">Ring Bell</button>
   <script>
      function youRang() {
         document.getElementById("main-text").innerHTML += "you rang...";
         console.log("you rang...");
      }
      // Obtain a reference to the button element
      let button = document.getElementById("ring-button");
      // Set named function youRang as the click event handler
      button.addEventListener("click", youRang);
   </script>
</body>
</html>

```

Result (if button is clicked)

```
affect on page: adds "you rang..." to <p>
output in console: you rang...
```

Warning

Be sure to use the correct event name when declaring the event name. An error will NOT be thrown if an invalid event name is given.

Note

This chapter uses DOM methods to add event handlers. When searching online, you may find examples using jQuery to add event handlers, which look like `.on("click", ...)` or `.click(...)`. **jQuery** is a JavaScript library designed to simplify working with the DOM. jQuery's popularity has declined as the DOM itself has gained features and improved usablity.
The second parameter of `addEventListener` is a function. Remember there are many ways to declare a function in JavaScript. So far, we have passed in named functions as the event handler. `addEventListener` will accept any valid function as the event handler. It's possible, and quite common, to pass in an *anonymous function* as the event handler.

```
anElement.addEventListener("eventName", function() {
  // function body of anonymous function
  // this function will be executed when the event is triggered
});

```

23.5.2. Event Details

A benefit of using `addEventListener` is that an *event* parameter is passed as the parameter to the event handler function. This event is an object instance of the Event class, which defines methods and properties related to events.

```
anElement.addEventListener("eventName", function(event) {
  console.log("event type", event.type);
  console.log("event target", event.target);
});

```

`event.type` is a string name of the event.

`event.target` is an element object that was the target of the event.

Try It!

Above, we saw how we could use `addEventListener` to add the function `youRang()` as the event handler for the `Ring Bell` button.

Using `addEventListener`, could you add the function `greetFriends()` as the event handler for the `Greet Friends` button?

Try it at repl.it

23.5.3. Event Bubbling

Remember that the DOM is a tree of elements with an `<html>` element at the root. The tree structure of an html page is made of elements inside of elements. That layering effect can cause some events, like *click*, to be triggered on a series of elements. **Bubbling** refers to an event being propagated to ancestor elements, when an

event is triggered on an element that has parent elements. Events are triggered first on the element that is most closely affected by the event.

Example

We can add a *click* handler to a `<button>`, a `<div>`, and the `<html>` element via the `document` global variable.

```
<!DOCTYPE html>
<html>
<head>
   <title>Event Bubbling</title>
   <style>
      #toolbar {
          padding: 20px;
          border: 1px solid black;
          background-color:darkcyan;
      }
  </style>
</head>
<body>
   <div id="toolbar">
      <button id="ring-button">Ring Bell</button>
   </div>
   <script>
      let button = document.getElementById("ring-button");
      button.addEventListener("click", function (event) {
          console.log("button clicked");
      });
      document.getElementById("toolbar").addEventListener("click", function (even
t) {
          console.log("toolbar clicked");
      });
      document.addEventListener("click", function (event) {
          console.log("document clicked");
      });
   </script>
</body>
</html>
```

Console Output (if button is clicked)

```
button clicked
toolbar clicked
document clicked
```

In some cases, you may want to stop events from bubbling up. We can use `event.stopPropagation()` to stop events from being sent to ancestor elements. Handlers for parent elements will not be triggered if a child element calls `event.stopPropagation()`.

```
button.addEventListener("click", function (event) {
   console.log("button clicked");
   event.stopPropagation();
});
```

Try It!

With the HTML above, what happens when you click in the green?

After you see the result, try adding `stopPropagation()` to the button click handler and seeing what happens when you click the button.

Try it at repl.it

23.5.4. Check Your Understanding

Question

Do these code snippets have the same effect? `button.addEventListener("click", youRang)` and `<button onclick="youRang();">`

Question

Can *click* events be prevented from bubbling up to ancestor element(s)?

Question

What is passed as the *argument* to the event handler function?

23.6. Event Types

DOM and JavaScript can handle numerous event types. We will discuss a few different types of events here. As you continue your studies of the DOM and events, you may find these two reference links helpful.

1. W3Schools Event reference
2. MDN Event reference

23.6.1. Load Event

The DOM includes the **load event**, which is triggered when the window, elements, and resources have been *loaded* by the browser. Why is it important to know when things have loaded? Remember you can't interact with HTML elements in JavaScript unless they have been loaded into the DOM.

Previously, we were moving the `<script>` element *below* any HTML elements that we needed to reference in the DOM. Using the *load event* on the global variable `window` is an alternative to `<script>` placement. When the *load event* has triggered on the *window* as a whole, we can know that all the elements are ready to be used.

Example

A `<script>` tag is in `<header>` and all DOM code is inside *load* event handler.

```
<!DOCTYPE html>
<html>
<head>
   <title>Window Load Event</title>
   <script>
      // add load event handler to window
      window.addEventListener("load", function() {
          // put DOM code here to ensure elements have been loaded
          console.log('window loaded');

          let ring = document.getElementById("ring-button");
          ring.addEventListener("click", function (event) {
              console.log("ring ring");
          });

          let knock = document.getElementById("knock-button");
          knock.addEventListener("click", function (event) {
              console.log("knock knock");
          });
      });
   </script>
</head>
<body>
   <div id="toolbar">
      <button id="ring-button">Ring Bell</button>
      <button id="knock-button">Knock on Door</button>
   </div>
</body>
</html>
```

Console Output (if "Knock on Door" button is clicked)

```
window loaded
knock knock
```

23.6.2. Mouseover Event

There are many mouse related DOM events. We have already used the *click* event. Another example of a mouse event is the **mouseover** event, which is triggered when the mouse pointer enters an element.

Example

We can use *mouseover* event to add a `">"` to the `innerHTML` of the element that the mouse pointer has been moved over.

```
<!DOCTYPE html>
<html>
<head>
   <title>Mouseover Event</title>
   <script>
         window.addEventListener("load", function() {
            let list = document.getElementById("lane-list");
            list.addEventListener("mouseover", function (event) {
               let element = event.target;
               element.innerHTML += ">";
               console.log("target", element);
            });
         });
   </script>
</head>
<body>
   Mouseover Race
   <ul id="lane-list">
         <li>Lane 1</li>
         <li>Lane 2</li>
         <li>Lane 3</li>
   </ul>
</body>
</html>
```

Example HTML Output (if the mouse is moved over elements in the list)

```
Mouseover Race

   Lane 1>>>>>>>
   Lane 2>>>>>>>>>>>>
   Lane 3>>>>>>>>
```

23.6.3. Check Your Understanding

Question

What error happens when you try to access an element that has not been loaded into the DOM?

Question

What is *true* when the *load* event is triggered on `window`?

a. The console is clear.
b. All elements and resources have been loaded by the browser.
c. Your files have finished uploading.

23.7. Exercises: The DOM and Events

Time to make a flight simulator for your fellow astronauts! The provided HTML and JavaScript files can be used for all of the exercises. For each exercise, the requirements and desired effect of the events is listed.

Repl.it with starter code

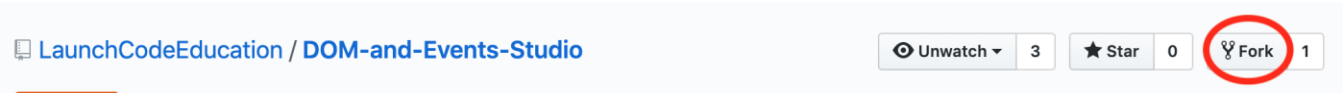
1. When the “Take off” button is clicked, the text “The shuttle is on the ground” changes to “Houston, we have lift off!”. The “Take off” button has an `id` of “liftoff”.
2. When the user’s mouse goes over the “Abort Mission” button, the button’s background turns red. The “Abort Mission” button has an `id` of “abortMission”.
3. When the user clicks the “Abort Mission” button, make a confirmation window that says “Are you sure you want to abort the mission?”.

23.8. Studio: The DOM and Events

Now that we can build a basic flight simulator, we want to add more controls for the staff at our space station. The HTML, CSS, and JavaScript files are provided. For each event, the requirements and desired effect is listed.

23.8.1. Getting Started

First, fork the studio repository to your Github account. To do so, on the studio repository page on Github, click the "Fork" button.



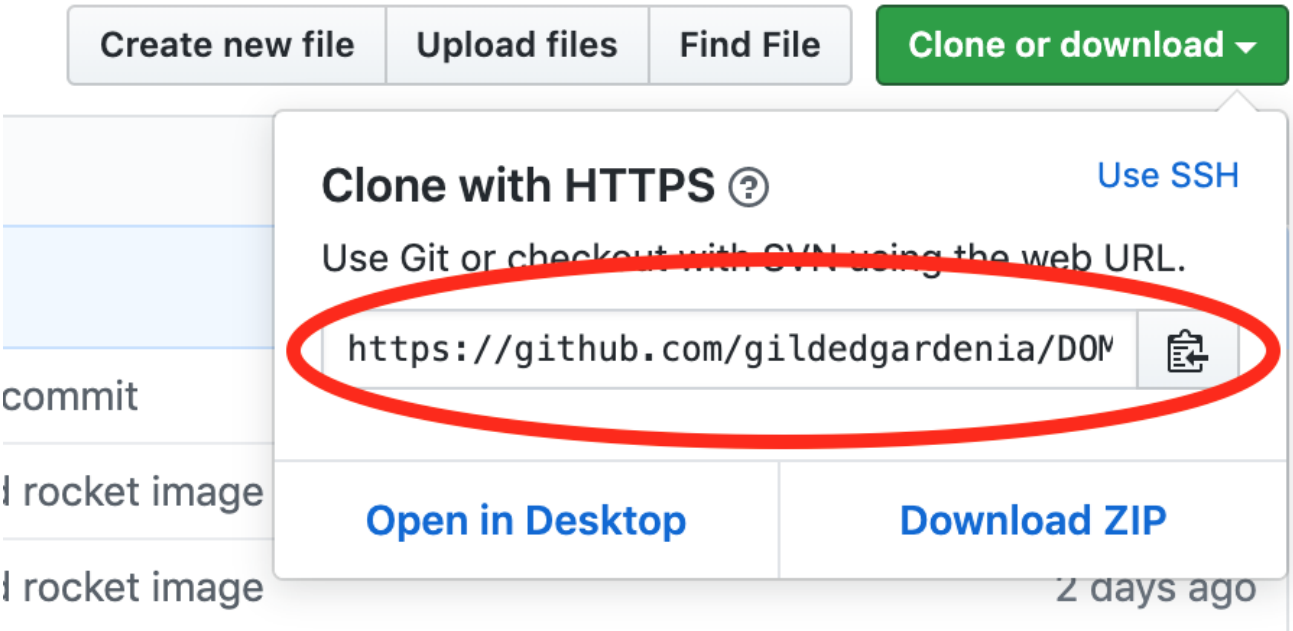
A popup appears asking where to fork the repository to and you select your profile. You should now have a copy of the repository on your own profile!

Note

Not only is forking repositories an important Git skill, it is especially vital in the class so that everyone has the same starter code! Before continuing, make sure that the repository is now on your profile and you are working with your copy of the starter code for the rest of the studio.

If you have properly forked a repository, when you click on the forked repository on your profile, you will see the following:

Once you have properly forked the repository, you can clone the remote repository to your computer. To start, click on the green "Clone" button to get the proper HTTPS url for the command.



Copy the url either by clicking on the button with a clipboard icon or highlighting the whole url and copying.

In the terminal, navigate to the directory where you want to put your new project. Use the command, `git clone <url>`, with the url you just copied to put the project on your local machine.

Note

`git clone` will clone a whole directory including the Git repository on your machine, so there is no need to initialize a Git repository in a new directory to get started this way!

Open Visual Studio Code and go to *File* > *Open* to find your new project and get started!

23.8.2. The Requirements

1. Use the window *load* event to ensure all elements have loaded before attaching event handlers.
2. When the "Take off" button is clicked, the following should happen:
    1. A window confirm should let the user know "Confirm that the shuttle is ready for takeoff." If the shuttle is ready for liftoff, then add steps 2-4.
    2. The flight status should change to "Shuttle in flight."
    3. The background color of the shuttle flight screen (`id = "shuttleBackground"`) should change from green to blue.
    4. The shuttle height should increase by 10,000 miles.
3. When the "Land" button is clicked, the following should happen:
    1. A window alert should let the user know "The shuttle is landing. Landing gear engaged."
    2. The flight status should change to "The shuttle has landed."
    3. The background color of the shuttle flight screen should change from blue to green.
    4. The shuttle height should go down to 0.
4. When the "Abort Mission" button is clicked, the following should happen:
    1. A window confirm should let the user know "Confirm that you want to abort the mission." If the user wants to abort the mission, then add steps 2-4.
    2. The flight status should change to "Mission aborted."
    3. The background color of the shuttle flight screen should change from blue to green.
    4. The shuttle height should go do to 0.
5. When the "Up", "Down", "Right", and "Left" buttons are clicked, the following should happen:
    1. The rocket image should move 10 px in the direction of the button that was clicked.
    2. If the "Up" or "Down" buttons were clicked, then the shuttle height should increase or decrease by 10,000 miles.

23.8.3. Bonus Mission

If you are done with the above and have some time left during class, there are a few problems that you can tackle for a bonus mission.

1. Keep the rocket from flying off of the background.
2. Return the rocket to its original position on the background when it has been landed or the mission was aborted.