Getting Stuff Done*

*with HTML and JavaScript

digIT 2019 winter web workshop

Quynh-Chi Nguyen

During the summer camp web workshop, we looked at HTML and CSS, which allowed us to place elements on a webpage and control how they look and where they are. We also looked at the DOM tree and how it reflects the structure of HTML.

In this workshop, we're going to look at HTML inputs, which allow users to interact with a page, and then use JavaScript to take user input and use it to manipulate the DOM tree. With JavaScript, you can write code that runs in (almost) any web browser, and you don't need any special tools to run or write it either.

We're going to use a tool called CodePen for this workshop. This lets you write HTML, CSS and JavaScript directly in your browser and you can see the changes as you go along without having to upload files or do anything special.

About JavaScript

JavaScript is behind most of the interactivity you see on websites, and more that you don't see! One of the best things about JavaScript is that it'll run in pretty much any web browser. You don't need to install extra programs to get it working.

JavaScript has some similarities to Python, which you've already used. They're both interpreted languages - you don't need to compile them before you can run them. In both languages, you don't have to assign a type to a variable (tell the program if a variable represents a number or text, for example). The interpreter figures it out from context.

It's also different in some ways. The most obvious one is code formatting. Python cares about indentation and whitespace (spaces and tabs). JavaScript works out which instruction is which using brackets, braces and semicolons. Of course, there are different keywords and functions available in each language, and things behave differently under the hood. But the way you'll think about writing code and breaking down the steps will be similar.

JavaScript is not related to Java! They just happen to have similar names.

Today's workshop

Remember the DOM tree from our summer workshops? It's the "tree" of all the elements in your HTML page. We'll still want to start with some HTML to build the DOM, but we can then use JavaScript to manipulate it on the fly, by adding and removing elements and changing their properties. We'll also want a way to trigger the manipulation, which is where events come in.

DOM events are sent by your browser to the code ("triggered") to say "hey, this happened! Do you want to do something about it?" They can be triggered on a wide range of events, including keyboard and mouse input, page load or unload, errors and more - you'll find a full list at **https://developer.mozilla.org/en-US/docs/Web/Events**. We're going to keep it pretty simple today and look at doing things when you click a button (a super common usage!) or tick/untick a checkbox. More specifically, we're going to build a simple to-do list, where you can add items and check them off.

As we've already seen, web development has all kinds of tracks; everyone here is at different levels and some people prefer to code, others to design, so you'll have the opportunity to choose what you want to delve into and work at your own pace, too.

We'll start together by looking at HTML inputs and some JavaScript basics, then revise the DOM tree and explore how to add nodes, remove nodes and move nodes around in the DOM tree using JavaScript. We'll then have a look at CodePen and today's prepared exercise.

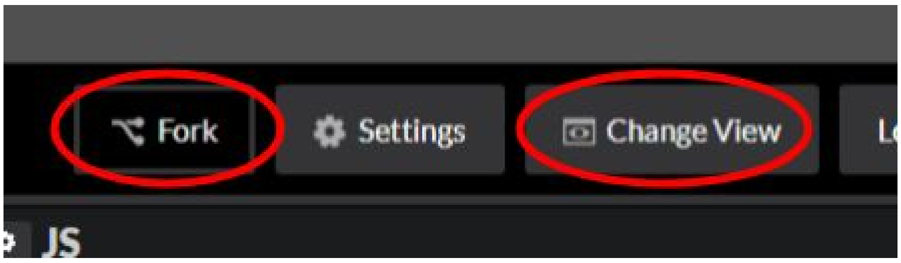
All the links mentioned in this workshop (and more) are available at **https://digit-winter.neocities.org/** - you can load that page to click on them instead of typing them into your browser. You'll also find a digital copy of this handout, a copy of the slides I'll use in this workshop, and the summer workshop handout there.

Setting up your CodePen environment

CodePen is what's called a code playground - it lets you edit HTML, CSS and JavaScript all in the same page, runs it for you, and takes care of importing libraries (other people's code that you might want to use) and setting everything up to work together so you can play with the fun stuff.

Here's the pen you'll need to get you started:

https://codepen.io/qcn/pen/XLZamN



Open up the pen in your browser. You'll need to fork it by clicking the "Fork" button at the top of the page. Forking makes your own copy that you can work on and save, without destroying the original. (In fact, you won't be able to save your work without forking, because the original wasn't created by you.)

You can make an account on CodePen if you want (don't give out any identifying information!), or just use it anonymously (in which case, make a note of the address in your browser, so you can come back to it later).

You'll probably want to change some settings to make it easier to work with the pen. I like to change the layout so my HTML, CSS and JavaScript editors are next to the content - click "Change View" at the top. Since we're working on a to-do list, it makes more sense to preview our content in a vertical space!

(I also prefer a dark-on-light colour scheme, rather than the default light-on-dark - if you sign up for an account, you can change the colour scheme in your settings.)

Option A: Code

The provided codepen has a skeleton that could one day be a to-do list. Your job today is to flesh it out to make it work! Keep in mind that you're by no means expected to finish the entire set of exercises in the short time we have today - the workshop is designed for you to get a taste and work at your own pace if you'd like to explore further. If all you do today is read through the booklet or play around on Grok or Neocities, that's fine!

Let's break some of this code down.

The HTML

```
<div class="container">
  <h1>Add item</h1>
  <input type="text" id="addItemInput" placeholder="I need
to...">
  <button type="button" onClick="addItem()">+</button>
  <hr>
  <h1>To Do</h1>
  <ul id="todoList">
  </ul>
  <h1>Done</h1>
  <ul id="doneList">
  </ul>
</div>
```

You've seen most of these elements several times by now: we have a few headings, a text input and a button, and a couple of (unordered) lists. The important things to note are the `id` attributes, which will allow us to easily *target* these elements in JavaScript, and the `onClick` attribute on the button. This sets up the button to run the JavaScript function `addItem` when clicked. We'll see this function in a bit.

The CSS

Again, you've seen all of this before; there's nothing super-interesting here, except that we're using CSS to cross off list elements that are specifically inside the `doneList` (if we applied the text-decoration to all `li` elements, it would cross off ones we haven't yet done!).

The JavaScript

Constants, Variables and Finding Things in a Page

```
const todoList = document.getElementById("todoList");
```

This code assigns a *value* to a *name* using the `const` keyword, which says that it's a constant whose value does not change (as opposed to a variable, which uses the `let` keyword in the same way). Unlike in Python, you have to explicitly *declare* a name using one of these keywords - you can't just assign a value to a name.

Notice the semicolon at the end of the line. In Python, a line is a line; the whitespace tells the interpreter that the command is finished and we're starting a new command on the new line. This isn't the case in JavaScript: a single command can span multiple lines, and we tell the interpreter we're done with a command by ending it with a semicolon. This way, we can also have multiple distinct commands on one line, by ending each command with a semicolon.

`document` represents the entire web page in our browser; it's a reference to the top level (*root node*) of the DOM tree. `getElementById` searches a node for an element with the given `id` attribute, so in this case, it's going to find our `ul` element with an `id` of `todoList`. So this line of code is going to search the HTML document for an element (of any type) with `id="todoList"`, and assign it the name `todoList` in our Javascript code, so we can refer to it later in the code.

The JavaScript Console

CodePen has a command-line console to JavaScript built-in, where you can run JavaScript commands and test out your code. In fact, all modern web browsers have their own development tools including a JavaScript console! You can open the CodePen console by clicking on the "Console" button at the bottom of the screen.

Try running the command `document.getElementById("todoList")` in the console. It should print out and return the `ul` element you're looking for. You can also assign values to constants and variables, do calculations and inspect objects - for example, running the line `document.getElementById("addItemInput").placeholder` will show you the value of the placeholder attribute for the `addItemInput` element. Feel free to spend some time experimenting with the console!

Exercise 1

Add a single line of code to create a constant named `doneList` that refers to the `ul` element with `id="doneList"` - this should look a lot like the line we just saw that defined the constant named `todoList`. (There's a comment showing you where to put your line of code.) We'll use this a bit later.

Act 1: Addition

Our simple to-do list needs three actions: adding an item, moving an item between the "to-do" and "done" lists, and deleting an item. We'll now fill in the code to do these actions.

```
function addItem() {
  const newItemText = document.getElementById("addItemInput")
                            .value;
  createNewItem(newItemText);
  clearInput();
}
```

This creates a *function* (a group of commands that **doesn't get run until the function is called**) named `addItem`, which calls two other functions, `createNewItem` and `clearInput`. The `function` keyword works just like the `def` keyword in Python, but again, the syntax is a bit different: we wrap the argument list in brackets (there are actually no arguments to this function, but we still need the brackets!), and the function body is wrapped in curly braces.

We break the action down into steps to make it easier to understand, plan and code - first we're going to make a new item and add it to the DOM tree, then we're going to clear the text input so that it's empty for the next item we want to add.

Exercise 2

We're creating a constant named `newItemText` with the value of `document.getElementById("addItemInput").value`. What does this mean? Try entering `document.getElementById("addItemInput").value` into the JavaScript console to see if you can work it out. Type different things into the text input and re-run the line of code.

```
 1 function createNewItem(itemText) {
 2   let newItem = document.createElement("li");
 3
 4   // Fill me in to create an input element called "checkBox".
 5   checkBox.type = "checkbox";
 6   checkBox.addEventListener("change", moveItem);
 7
 8   // Fill me in to create a label element called "itemLabel".
 9   itemLabel.appendChild(document.createTextNode(itemText));
10
11   let deleteButton = document.createElement("button");
12   // Fill this line in to add some text to the button.
13   // Fill this line in to add an event listener that
14   // calls "deleteItem" on the "click" event.
15
16   // Fill me in to put all the elements we just created
17   // together into a new node for our DOM tree.
18
19   todoList.appendChild(newItem);
20 }
```

In order to create the new list item and put it on the page, we're going to build up a new node for our DOM tree, then attach it in the right place. The very first and last steps have already been done for you - we need to fill in the middle! I've numbered these lines so we can fill in different bits of this function. Lines starting with `//` are *comments* in JavaScript - anything after the `//` is ignored by the browser.

As with `addItem`, we're defining a function here. The main difference is that we are now passing an **argument** named `itemText` into the function (line 1) - *inside* the function, we can refer to `itemText` like any other variable, but *outside* this particular function we wouldn't be able to access it. (This concept is called *variable scope*, and deals with what variables can be accessed at a given location in code.)

Line 2 creates an `li` element in the document and calls it `newItem`. This is going to be our node containing all the information about our new to-do list item. At this point, we've created the element, but we haven't attached it anywhere into the tree - it's kind of just floating around. We'll need to attach it at some stage, so the browser knows where to show it on the page.

Exercise 3

- Use the same technique as on line 2 to replace line 4 with a line of code that creates a new variable called `checkBox`, which should initially be an `input` element. We're going to add some properties to this element and add it inside our list item.
- While you're at it, replace line 8 with a line of code that creates a `label` element in a variable called `itemLabel`.

As we saw earlier in today's workshop, `input` elements can have different types. In order to make this input appear as a checkbox instead of a text box, Line 5 sets the `type` property of the `checkBox` element to `checkbox`.

We're also going to want to move the attached list item when a checkbox is ticked or unticked. We'll worry about the exact details of *how* to do that a bit later; right now, we just want to tell the browser that when we *change* the checkbox by checking or unchecking it, we want to *call* the function named `moveItem`. We do this by adding an *event listener* - the browser will "listen" for the change "event" to happen, and when it does, it'll fire off a call to the specified function. This happens on line 6.

Line 9 creates a text node as a child of the label node. A text node is what contains the actual text within an element - even though it doesn't have HTML tags around it, it's still its own node within the DOM tree. We want to create a text node containing the `itemText` we passed into the function, and put it into our DOM tree as a label we just created, so that the label will contain the text we want.

Exercise 4

- Write some code on line 12 that creates a text node and adds the text to the delete button, like line 9 in the example.
- Then write some code on line 13 to call the `deleteItem` function (again, we'll fill this in later) on the click event of the delete button, like line 6. Notice that the checkbox and button have different events - not all events make sense on all elements!

Remember how we said we'd have to attach the `newItem` node to the DOM tree at some stage? The last pre-filled line is going to insert our new list item node into the DOM tree as a child of the to-do list node using the built-in `appendChild` function, but first, we have to put all the nodes we created together! We'll need to make `checkBox`, `label` and `deleteButton` children of `newItem` - preferably in that order.

Exercise 5

- Use the `appendChild` function to add `checkBox`, `label` and `deleteButton` (in that order) as children of the `newItem` node. You'll probably want to do this as 3 separate lines.
- You might want to put a space in between the label and the delete button, so they don't stick right up against each other - how might you do that?

This completes the first step of adding a new list item.

When we click the add button, we also want to clear the value from the text input box. We figured out that `.value` shows us the text currently entered into the text input, but reading from that property isn't the only thing we can do with it - we can also write to it! We can assign a value to it using =, the same way we do with any other variable.

Exercise 6

Fill in the `resetInput` function to set the `value` of the text input back to `""`, the empty string.

Act 2: How Moving!

Now we need a way to move list items between our two separate lists. We already set up our checkbox to move an item when it gets ticked or unticked; we just have to fill in the rest of the code to do it!

```
function moveItem() {
  const listItem = this.parentNode;
  if (this.checked) {
    // Fill me in!
  }
  else {
    // Fill me in!
  }
}
```

The `this` keyword represents the element the event triggering the function comes from. In this case, since we are calling the function by clicking on our checkbox, this will represent the checkbox element. The `parentNode` property gives us the parent node of an element. Recalling that our checkbox node is a child of the list item, we can get a reference to the list item element this way, and assign it to the constant named `listItem`.

This flow looks like an if/else flow in Python - it just has different syntax. The *condition* has to be wrapped in round brackets. Each *block* is wrapped in curly braces. Remember, lines of code end with semicolons.

We could also write the condition as `if (this.checked === true)`. In Javascript, we have both `===` and `==` . This may seem a little confusing, but you'll almost always want to use the triple equals - the double equals tries to convert the two sides to the same type of data before comparing them, while the triple just compares them. The equivalent inequality operator is `!==` (the double equals matches `!=`). Leaving out the `=== true` defaults to checking if the condition is "truthy", which can be pretty confusing in Javascript.

Exercise 7

Decide which list the item should be in if the checkbox is ticked, and which list it should be in otherwise. Then use the `appendChild` function we saw earlier to fill in the marked lines and move the *entire* list item to the right parent node. (Remember how we defined variable names for `todoList` and `doneList` before? You'll probably need these as parents now.) We don't need to remove it from where it is - the `appendChild` function will detach the node from wherever it currently is in the tree, along with all its children (like breaking a branch off a tree), and re-attach it in its new spot (like grafting it back on).

Act 3: Deletion

We're almost there! Now we just have the `deleteItem` function to finish.

Exercise 8

Fill in the `deleteItem` function to remove a list item from the list it's in. You'll need the following:

- You'll want to get the list item element in the same way as `moveItem`.
- The `remove()` function will remove an element from the DOM tree completely.

Congratulations - you now have a functional to-do list that will let you add, delete and check off items. Now you can try the design task, or the optional challenge below - or find some other way to improve your list!

Here’s one I prepared earlier

I’ve made another CodePen implementing all the suggestions from this handout for you to look at. Have a look at this one later in your own time, or right at the end of the workshop - see what you can do on your own first.

https://codepen.io/qcn/pen/wLyqaO

There are extensive comments in this pen giving details of what it’s doing. You might not understand it all straight away - that’s okay! See if you can work through it, figure out what it’s doing, change things to see what they do, fork it and add to it yourself!

Optional Challenge

Add an edit button to your to-do list items. This is a bit more complex, and while I’ll help to break it down into steps, you’ll need to use other references to help you figure out exactly *how* to do each step.

You’ll need to:

- Create an “edit” button and add it to the list element, like the delete button.
- Find the current text of the to-do list item. *Hint*: look at the `innerText` of the label inside this list element. You can use the JavaScript console to inspect the value.
- Create a text input, containing the same text (because you want to edit the current item, not replace it completely).
- Create a save button for the input.
- Replace the contents of the list element with your input and save button.
- Make the save button replace the contents of the list element with a to-do list item again. *Hint:* as a first step, it’s probably easiest to use the existing delete and create functions to delete the current item and add a new one! Afterwards, you can think about how to replace the existing item, so they stay in the same order.

Resources

- Mozilla’s JavaScript tutorials and references: **https://developer.mozilla.org/en-US/docs/Web/JavaScript**
- Mozilla’s HTML tutorials and references: **https://developer.mozilla.org/en-US/docs/Web/HTML**

Option B: Design

Remember, web development doesn't all have to be about making your site *do* things - you can make it look good too! If you'd rather play around with CSS than write JavaScript, here's a codepen I prepared earlier that's already pretty functional:

https://codepen.io/qcn/pen/wLyqaO

The provided CSS looks super basic and ugly. Fork the codepen (see page 3 of this handout) and try using CSS to do some of the following, or anything else you can think up! Feel free to edit the HTML if necessary, to add classes, IDs or containers.

- Play around with the spacing and positioning of each area - come up with something that looks good to you!
- Remove the dot points from each item - the checkboxes already make them look like list items! Try to do this without changing the HTML - we still want them to be in `<li>` tags (remember the concept of semantic HTML, where the tags you use reflect the purpose of the part of text you're marking up).
- Add some kind of background - lined paper might be nice for a list? As a bonus challenge, you could try to do this without an image, just using CSS! (Hint: you'll want to consider CSS gradients, and how you could make a gradient that looks like just lines on a page instead of gradually changing over a larger area.)
- Change the fonts used. Remember, Google Fonts has a great selection of fonts you can include for free!

Resources

- **Grok Learning** - it's always a good idea to revise previous knowledge, or work through exercises you didn't get to earlier.
- Mozilla has fantastic CSS documentation in addition to its HTML and Javascript resources: see **https://developer.mozilla.org/en-US/docs/Web/CSS** for tutorials, articles and a comprehensive reference.
- CSS-Tricks has a great reference in its Almanac section, as well as examples, snippets and articles. Some of this content will be a bit more advanced than we're ready for right now, but knowing where to look in the future is always useful! **https://css-tricks.com/**
- Your booklet from summer camp has a CSS quick-reference/cheatsheet included; if you don't have a copy anymore, you can grab a PDF from **https://digit-winter.neocities.org/**.