Programming

It’s time to unlock what JavaScript can do when it’s not just a calculator. We’ve skipped over a lot of details about data types to get to here, but it’s important to start thinking *programmatically* as soon as possible. Rules and properties we can always look up. How to do things programmatically, however, is a skill that needs to be nurtured. As a result, it’s a source for early frustrations as well. Just remember, programming is puzzle solving, and in this part of the process you can finally start thinking of the puzzle as a journey.

## Control flow

**Control flow** is an idea you’ve probably seen before, like in flow charts. They’re all over social media and often funny. In a flow chart, you start from some position and answer questions. Depending on the answers to those questions, you end up in a certain location. Other answers lead you somewhere else. But the idea is that you are interacting with information, in that you are being provided a prompt for some input, and your input directs what happens.

Let’s sketch out a toy program to begin illustrating control flow in a program.

We want to write a program that asks the user what they want for dinner. If they answer “burrito,” the program congratulates their choice. If they answer anything else, the program scolds them for not wanting a burrito. What might that look like in **pseudocode** (pretend programming that’s not a real language)? Let’s try it out while also using some JavaScript we already know, like let and console.log().

First, the program needs to get the information from the user, so we need some kind of input. Let’s save that as a variable.

// THIS IS PSEUDOCODE. It is for illustration only. It will crash.  
//  
let userInput;  
userInput = prompt\_the\_user\_for\_what\_they\_want\_for\_dinner;

We have a variable now, userInput, that has whatever the user has input. Now let’s test that variable, using the === operator you’ve already learned.

// THIS IS PSEUDOCODE. It is for illustration only. It will crash.  
//  
if userInput === "burrito";  
 then console.log("Brilliant choice!");

OK. But what if the input *isn’t* “burrito”?

// THIS IS PSEUDOCODE. It is for illustration only. It will crash.  
//  
if userInput !== "burrito";  
 then console.log("Don’t you want a burrito?");

Here I’m using the negation operator !==. It’s the same as ===, but its inverse.

And that’s it. We have our program. The JavaScript, as we’ll see, isn’t so terribly different from this code we already have.

## If statements

An if statement is an example of **conditional statement**. That means that it behaves in a certain way depending on a condition. From the example above, we can read

// THIS IS PSEUDOCODE. It is for illustration only. It will crash.  
//  
if userInput === "burrito";  
 then console.log("Brilliant choice!");

As “If the condition that the variable userInput is equivalent to the string ‘burrito’ is true, then respond to the console the string ‘Brilliant choice!’”

That’s a mouthful, but it is actually three distinct steps:

1. The test of whether userInput is equivalent to the string “burrito.”
2. If the test in 1. is true, then the condition in the if statement is met, so we can go on to 3.
3. log to the console the string “Brilliant choice!”

Now, JavaScript’s syntax is different from the pseudocode above. First, there is no then statement. Instead, that then is replaced with braces ({}). Everything within the braces gets executed if the truth test (userInput === "burrito") returns true. And the truth test itself is surrounded by parentheses. Generically, then, it looks like this:

// This code is for illustration only. It will crash your console.  
//  
if ( truth\_test ) {  
 do\_things\_if\_the\_test\_returns\_true;  
}

Note where the semicolons are (and are not!) in this example. Now we can fill it out with some of our toy program. We know that do\_things… is actually console.log("Brilliant choice!"), so we can put that in. As for truth\_test, let’s just put in true for now.

> if ( true ) {  
 console.log("Brilliant choice!");  
 }  
//--> Brilliant choice!

Type this into the console (pressing the return key where appropriate). You should get “Brilliant choice!” logged to your console. Now replace true with false. What happens? Why?

Now you can add some complexity, this time using the variable userInput:

> let userInput;  
> userInput = "burrito";  
> if ( userInput === "burrito" ) {  
 console.log("Brilliant choice!");  
 }  
//--> Brilliant choice!

Notice where you have to use = (which *assigns* a value to a variable) and where you have to use === (which *tests* whether something is true). If you type this in the console, again it should congratulate you. If you replace the second line with userInput = "samosa";, what happens? Why?

In our program, we had a second condition, which would scold the user if they didn’t want a burrito. That’s pretty straightforward to write:

> let userInput;  
> userInput = "samosa";  
> if ( userInput !== "burrito" ) {  
 console.log("Don’t you want a burrito?");  
 }  
//--> Don’t you want a burrito?

However, we can *join* both truth tests using else:

> let userInput;  
> userInput = "samosa";  
> if ( userInput === "burrito" ) {  
 console.log("Brilliant choice!");  
 } else {  
 console.log("Don’t you want a burrito?");  
 }  
//--> Don’t you want a burrito?

Now, we can see this as “if the truth test is true, then log ‘Brilliant choice!’ Otherwise, respond ‘Don’t you want a burrito?’” Less typing! If you type this snippet into your console, you’ll see that it scolds you. What do you have to change so that it congratulates you?

So far so good, but we still have the line of the pseudocode, userInput = prompt\_the\_user\_for\_what\_they\_want\_for\_dinner;. Getting that part to work requires writing some HTML.

## Embedding JavaScript in a webpage

Typing things into the console becomes tedious to have to retype everything every time. Fixing mistakes is difficult. And if you close the window (or reload your browser), you lose everything you’ve already done. There must be a better way to write and execute JavaScript, right? Of course there is.

Back in [Chapter one](/1-environment/), I had you clone a project with Atom, and that project included a file called index.html. Open that file up in Atom by double clicking on it in the Projects tab on the left of the Atom window. Now we need to open this file in a browser, as well. This is a bit tricky, but if you look at the title bar for Atom, it should say something like “Project — ~/github/javascripting-english-major-project.” That means the file is in a folder, called javascripting-english-major-project. That folder, in turn, is in a folder called github, which is in your home folder. On Windows, it’s saved somewhere else.

Open up your home folder, like you would to open up any kind of regular folder, and you should see the github folder. Open that up, and you should see a folder with your repository. If you open that up, you’ll see the file index.html. Double click on that, and it should open up in your browser. It’ll be boring, but it’ll be there. You should be greeted with a blank white webpage with, in large text, “This is my project!”

For the rest of this course, we will be using Atom *and* a web browser, so it’s time to get used to having both open at once. Back in Atom, change the text “This is my project” to something else and save your change. Switch over to the browser and press the reload button. The text should change.

Congratulations, you are now a web editor.[[1]](#footnote-25) Let’s make you a web developer, though. That involves adding a few lines to index.html, so go back to Atom.

Inside the <body> tags, underneath the <h1>, add:

<div id="response">This is the #response div.</div>  
<script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>  
<script src="scripts.js"></script>

Type that second line carefully. When you’re done, the whole of index.html should look like this:

<!doctype HTML>  
<html lang="en">  
 <head>  
 <meta charset="utf-8">  
 <title>My JavaScripting English Major Project</title>  
 </head>  
 <body>  
 <h1>This is my project!</h1>  
 <div id="response">This is the #response div.</div>  
 <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>  
 <script src="scripts.js"></script>  
 </body>  
</html>

Save and commit, like you learned earlier. A good commit message would be “Add jQuery and local JavaScript.”

There are two new tags here, <div> and <script>. The first is a generic container tag that lets you mark off a part of a page for content. Note that it has an id **attribute**, and the value of that attribute is response. Throughout, I’ll be referring to this <div> container as #response. The second tag, <script>, tells the webpage to look for files indicated by the src, or **source**, attribute. In other words, first it goes to a web server, code.jquery.com, and downloads a source file from there, called jquery-3.2.1.min.js. Then it looks for a local file, on your computer, called scripts.js. But that file does not yet exist. So now you can make it.

Back in Atom, right-click (or control-click) on your project icon and choose “New File” from the dropdown menu. A tiny window asking for the file’s name will appear, and you should type in scripts.js. In the new window that opens, type in:

alert("scripts.js has loaded!");

Save the file and reload the browser. You should get an alert, telling you that “scripts.js has been loaded!” If you do, stage and commit. A good commit message now would be “Create scripts.js.” In the project pane in Atom, you should see scripts.js alongside index.html.

That’s obviously pretty annoying, so change alert in scripts.js to console.log. Now open up the console on the browser and reload the page. As you can see, instead of typing JavaScript directly in the console, we can type it in Atom, instead, and reload our page. But even writing to the console isn’t terribly interesting, except when you’re testing or debugging. Let’s use JavaScript, instead, to write to the webpage *itself*.

There are a lot of ways to do that, but that <script> line about “jquery” above will make things a bit easier. [jQuery](http://jquery.com) is a powerful JavaScript **library**, meaning it is a set of tools that make programming easier. It lets us developers do many tasks succinctly and clearly. Replace your single line in scripts.js with this, then:

$("#response").html("scripts.js has loaded!");

Now reload the page in the browser. That earlier text, that read “This is the #reponse div.” is now replaced. We’ll learn jQuery in greater detail as we move along, but that one line of JavaScript does this:

* $(: Select something in the webpage with jQuery.
* "#response"): In fact, select the HTML element with the id of response. (The # means are looking for something with a specific id.)
* .html(: Do something with the HTML that is inside the element we selected.
* "scripts.js has loaded!");: Change the HTML with this new HTML.

The [jQuery selector](https://api.jquery.com/category/selectors/), $(""), is one of the most important bits of code on the web.[[2]](#footnote-28)

Catch your breath. A whole lot has happened in this section. Let’s return to the toy program from the previous section, though, and update the console.log() parts with the jQuery. In other words, make your scripts.js file look like this:

let userInput;  
userInput = "samosa";  
if ( userInput === "burrito" ) {  
 $("#response").html("Brilliant choice!");  
} else {  
 $("#response").html("Don’t you want a burrito?");  
}

Save and reload in the browser. What happens now? What if you change “samosa” to “burrito” in scripts.js, save, and reload?

We’re back to where we were at the start of this section, but now instead of logging information to the console, you’re changing text on the webpage. Yet there’s still that little detail about how to ask the user what they want for dinner.

In other words, we want to change userInput = "samosa"; in scripts.js to userInput = prompt\_the\_user\_for\_what\_they\_want\_for\_dinner;, but in JavaScript, not pseudocode.

There are a lot of ways to get information from the user, but for now we can use JavaScript’s prompt() function, which asks the user to type something in. This works on most browsers, but some (like Brave) might have it disabled. Change the second line of scripts.js to:

userInput = prompt("What do you want to have for dinner?", "Type your answer here.");

Save and reload. If all goes well, when you reload the page in the browser, you should immediately have a teeny window asking you what you want for dinner. No matter what you type in, unless it’s “burrito,” the webpage will scold you. If that works, commit.

You’ve written a full program. It takes in input, feeds it through control flow (in this case, a conditional statement), and delivers output appropriate to the input. Of course, it’s still pretty basic, but baby steps. Baby steps.

## While and for loops

If you’ve got conditionals down, we can now move to looping. Remember, in JavaScript, a conditional statement takes the form of:

// This code is for illustration only. It will crash your console.  
//  
if ( some\_condition\_is\_true ) {  
 do\_things;  
}

Loops use the same syntax:

// This code is for illustration only. It will crash your console.  
//  
while ( some\_condition\_is\_true ) {  
 do\_things;  
}  
  
for ( complicated\_stuff\_we’ll\_get\_to ) {  
 do\_things;  
}

### While loops

While loops keep executing over and over until the truth test becomes false. As a result, something like:

while ( 0 < 1 ) {  
 console.log("Zero is less than one.");  
}

will keep running forever or until your browser crashes. Whichever comes first. As you can see, the while loops can be dangerous if the truth test never stops being true. An if statement gets executed once and moves on. Loops, though. Well, it’s in the name!

Yet it’s possible to use while loops with a bit of discretion. For example, replace the contents of scripts.js with this:

let i;  
i = 1;  
while ( i < 4 ) {  
 $("#response").append("<br />" + i);  
 i = i + 1;  
}

Before saving and reloading the webpage, what do you think this does? Now when you reload the page and see what it does, can you figure out why? If yes, then looping already makes sense to you, and you’re a step ahead. If no, I’ll walk through what’s going on here step by step:

let i;  
i = 1;

First we define a variable i and assign it the number 1. It’s tradition in programming to call looping variables i, so I’m continuing that tradition.

while ( i < 4 ) {

Read that out in English: “while the variable i is less than the number four.” That means that as long as i is less than 4, the loop will be true, and the program will loop.

$("#response").append("<br />" + i);

This line is a bit sneaky. First, it uses .append() instead of .html(). That just means that it adds the HTML to the end the already existing HTML instead of replacing all of it. Next, it uses <br />, which is the HTML tag for making a line break. But note that little i at the end. We’ll come back to it in a second, but for now it should be clear that this line means “append the HTML line break tag and the value of the variable i to the HTML element #response.”

i = i + 1;  
}

Finally, and this is what gives the loop its magic, we see this expression. In plain English, this means, “assign to the variable i the value of the variable i plus 1.” This may seem like nonsense. After all, you can’t have “*x* = *x* + 1” in algebra class, which is probably the last time you dealt with variables. But in programming, this is allowed. And, in fact, it’s super useful.

Let’s look at the program in its entirety again:

let i;  
i = 1;  
while ( i < 4 ) {  
 $("#response").append("<br />" + i);  
 i = i + 1;  
}

Now we can see that the third line can be simplified, in English, to “append the HTML line break tag and ‘1’ to the #response HTML element.” Then the fourth line, in English, is “assign to the variable i the result of 1 + 1.”

At the end of the first time through the loop, then, i is now 2, not 1. Now we can see it go back to the beginning of the loop. First, truth test: is 2 < 4 true? Yes. Next, append a line break and “2.” Next, reset i to be equal to 2 + 1, or 3. Once back through the loop... Now reset i to be equal to 3 + 1, or 4 and…

Break. Because 4 < 4 is not true, the loop stops executing. More colloquially, we **break out of the loop**. And that’s why your webpage should read:

This is the #response div. 1 2 3

What are some changes we could make if we wanted it to go through the loop four times? Two should be immediately obvious, and both involve changing the truth test. We can change it either to i < 5 or i <= 4. Remember, the <= operator is the same as ≤, or less-than-or-equal-to.

It’s important to understand why this looping works the way it does, so it’s useful to loop over (as it were) this section until it’s clear.

### For loops

For loops are just like while loops, except the truth test is replaced by a three part expression relating to a **control variable**. The three parts are:

1. **Initialization**. What is the initial state of the control variable before the loop begins?
2. **Condition**. What is the truth test that the control variable has to pass?
3. **Afterthought**. What is the change made to the control variable each time through the loop?

If we look back at our while loop and consider i to be the control variable, we can see that a for loop is just a fancier version of a while loop. After all, i = 1; sets the **initial** state of the control variable before the loop begins. Next, i < 4 is the **condition** that the control variable has to pass in order for the loop to continue. Finally, i = i + 1; is the **afterthought**, or the change the control variable undergoes each time through the loop. In other words, you can collapse a five-line program into just three:

for (let i = 1; i < 4; i = i + 1 ) {  
 $("#response").append("<br />" + i);  
}

Note where the semicolons are placed, and also note that the two initialization lines

let i;  
i = 1;

are collapsed into one statement, let i = 1;. This works fine even outside the while loop, but it’s tidier to define your variables and assign them separately.

In addition to being terser, this syntax limits the control variable (i) to the loop itself, instead of defining it outside of the loop, like in a while loop.

Loops are tricky, but they’re vital to understanding how programming works. Try out the exercises to see how well you understand them.

## Exercises

(These are all to be written and tested using your project and the web browser.)

1. Write a program so that when you reload your page, it asks for a number and prints all the numbers from 1 to it, including it.
2. Building on that program, create it so that instead of printing the number, it prints something like:

This is the #response div. 1 is odd. 2 is even. 3 is odd. 1. Building on the previous program, have it print the same, except without that ugly “This is the #response div.” line. 1. Write down what seem to be common mistakes you are making. Are you forgetting to add some aspect of the JavaScript syntax?

## Footnotes

1. Yes, you are now writing HTML without learning how to do it. The key grammar of the markup is clear from this example, though. HTML is made up of nested tags that look like this, for example: <h1> to open and </h1> to close. Some tags, like the <meta> and <!doctype> tags don’t need to be closed, but most do. [↑](#footnote-ref-25)
2. To give a sense of jQuery’s pithiness, the same line in vanilla JavaScript would be: document.getElementById("response").innerHTML = "scripts.js has loaded!"; To my eyes, this is unnecessarily verbose. [↑](#footnote-ref-28)