Abstraction

Between strings, numbers, booleans, functions, arrays, and Objects, we have the fundamentals of JavaScript down. These are the pieces that build together to make all JavaScript web projects. But knowing how to put the pieces together is where programming stops being a series of clunky, step-by-step instructions and becomes creative expression.

Recall the example from [chapter 4](/4-functions/) of teaching the computer to make a burrito. One way to do it could be to list out every command, step by excruciatingly small step. Think about step 1: “get a tortilla.” What is a tortilla? What does it mean “to get”? Well, it seems that’s not step 1. Because there are steps that come before, that define both of those things. Abstraction is when you start combining these little chunks of code into something more elegant, something that reads more like English.

Furthermore, abstracting also leads to reusable code. Why not define a function get() in getting a tortilla, that you can then reuse for getting beans, for getting cheese, for getting guacamole? Define it once, reuse it forever.

Another way of thinking about abstraction is through a programming philosophy called **DRY**, for “Don’t Repeat Yourself.” If you find yourself writing the same kind of code over and over, it means you haven’t thought about the problem abstractly enough to realize moments where your code could benefit from abstraction.

If this all sounds abstract (as it were), we’ll get to some details in a bit. However, first we need to look back at functions and methods and learn an important detail I left out.

## Returning

In the [previous chapter](/5-collections/), when working with array methods, I asked you to think about why .push() and .pop() change the array, while .sort() and .reverse() do not. The why of the question is for discussion (and hopefully was already discussed in class), but now we have to think about the *how*.

Open up the JavaScript console, and type in let one; one = 1;. When you hit return, the console should read undefined. But now just type one; and hit return. Now the console responds with 1, which is what we assigned to the variable one.

In these two examples, undefined and 1 are the **return values** of the two commands you send the console. Every statement in JavaScript has a return value. The default is undefined, but with functions, you can set your own return value using the return keyword. For example, we have been using console.log() to write to the console so far, but try this in the console:

> let f;  
> f = function(){ return "I am a return value." };  
> f();

First, note that I collapsed the entire function definition to one line. But, second, note that the function prints text to the console. Let’s build on this by writing a function that combines someone’s first and last names into a full name with a space between them:

> let makeFullName, hughessFullName;  
> makeFullName = function(firstName, lastName){  
 firstName + " " + lastName;  
 }  
> hughessFullName = makeFullName("Langston", "Hughes");  
> console.log("Is your name " + hughessFullName + "?");  
//--> Is your name undefined?

The function is doing stuff, but we don’t get the result we want. We want the console to ask, “Is your name Langston Hughes?” not “Is your name undefined?” The problem is that the variable, hughessFullName, is assigned to undefined. But we want, instead, for that variable to hold the value "Langston Hughes". We do this by telling the function to **return** the string.

> let makeFullName, hughessFullName;  
> makeFullName = function(firstName, lastName){  
 return firstName + " " + lastName;  
 }  
> hughessFullName = makeFullName("Langston" "Hughes");  
> console.log("Is your name " + hughessFullName + "?");  
//--> Is your name Langston Hughes?

Knowing the return value of a function helps you manipulate it with confidence. For example, we know that .sort() and .reverse() *return* new arrays, leaving the original array unchanged. Since we know this, we can even **chain** the two methods:

> let turtles, sortedReversedTurtles;  
> turtles = ["Leonardo", "Donatello", "Raphael", "Michelangelo"];  
> sortedReversedTurtles = turtles.sort().reverse();  
//--> ["Raphael", "Michelangelo", "Leonardo", "Donatello"]

Say we accidentally included Splinter in the list of turtles, and decided to .pop() him off before reversing:

> let turtlesWithSplinter, reversedTurtlesWithoutSplinter;  
> turtlesWithSplinter = ["Leonardo", "Donatello", "Raphael", "Michelangelo", "Splinter"];  
> // oops. let's pop() Splinter off before reversing…  
> reversedTurtlesWithoutSplinter = turtlesWithSplinter.pop().reverse();

Uh-oh. This causes an error. Why does this happen? After all, we're just telling JavaScript to pop off the last value of the array, then sort it, and then reverse it. The answer is that, although .sort() and .reverse() return *arrays*, .pop() does not. It returns the *value* of the popped off element in the array. In other words, we're asking JavaScript to run .reverse() on "Splinter", which is a string. And strings have no .reverse() method. Error ensues.

Return values encourage programmers to think in terms of the effect of the way their functions manipulate data. This is valuable when we start to talk about iteration.

## Iterating

We know that strings can behave a bit like arrays, in that they have a .length property and index values. They also have the method .toUpperCase(), which makes a string all caps. What if we wanted to write a function that made every letter upper case *except* “e”? Let’s close the console for now and work this out in Atom. Erase your scripts.js and type in this:

let userString, upperCaseMinusE, upperCasedString;  
// First, we need a string from the user.  
userString = prompt("What do you want to UPPeRCASe?");  
// Second, we need to create our function.  
upperCaseMinusE = function(string){  
 // Something will happen here…  
};  
// Third, we need to pass the user’s string to the   
// function and assign the return value to a   
// variable.  
upperCasedString = upperCaseMinusE(userString);  
// And we can then print the string to the webpage.  
$("#response").html(upperCasedString);

Perhaps the easiest way to write the function would be to have it upper case the whole string at once and then just replace every “E” with “e.” But say the rules were that if a user *enters* an “E,” like “uppercase everything but the little ‘e’s and leave this ‘E’ alone,” then it wouldn’t work. Instead, let’s go over the string, letter by letter, and uppercase each letter on its own, while skipping the letter whenever it is “e.” That should be easy enough to write:

upperCaseMinusE = function(string){  
 if ( letter === "e" ) {  
 result = letter;  
 } else {  
 result = letter.toUpperCase();  
 }  
};

This code won’t yet work, mostly because it references two variables, letter, and result that are undefined. But the mechanics should be clear. Given a variable letter, if it’s equal to “e,” let it be and set result to it. If it’s not, make it upper case. Now, how do we iterate over it? We use a for loop. Remember, a for loop takes three parameters: the initialization, the condition, and the afterthought. So let’s add a for loop to our function:

upperCaseMinusE = function(string){  
 for ( let i = 0; i < string.length ; i = i + 1 ) {  
 if ( letter === "e" ) {  
 result = letter;  
 } else {  
 result = letter.toUpperCase();  
 }  
 }  
};

letter still isn’t defined, but we’re at least iterating over the string. Notice that the condition is that our counter variable, i, be less than the length of the string, which we get by asking for the string.length. But why not set i to 1, and make the condition i <= string.length? The answer has to do with zero indexing. Recall that to get the first letter of a string, we need to ask it for string[0]. So the first time through the loop, we want i to be 0, so that we can ask for string[i] and get the zeroth letter in the string. And that’s what we’ll assign to letter!

upperCaseMinusE = function(string){  
 for ( let i = 0; i < string.length ; i = i + 1 ) {  
 let letter;  
 letter = string[i];  
 if ( letter === "e" ) {  
 result = letter;  
 } else {  
 result = letter.toUpperCase();  
 }  
 }  
};

i, of course, is defined for the course of the loop, and it increases every time through it. As a result, we get access to each letter in our string variable by calling string[i] on it every time through the loop, when i has a different value. But that result variable is still undefined, and it’s still not doing anything. result will be what we return from the function. And we know that what we return will be a string, so let’s define result at the beginning of the function and assign it to a blank string. scripts.js should now look like this:

let userString, upperCaseMinusE, upperCasedString;  
userString = prompt("What do you want to UPPeRCASe?");  
upperCaseMinusE = function(string){  
 let result;  
 result = "";   
 for ( let i = 0; i < string.length ; i = i + 1 ) {  
 let letter;  
 letter = string[i];  
 if ( letter === "e" ) {  
 result = letter;  
 } else {  
 result = letter.toUpperCase();  
 }  
 }  
 return result;  
};  
upperCasedString = upperCaseMinusE(userString);  
$("#response").html(upperCasedString);

At last, the code isn’t broken any longer, so you can save, commit, and reload the webpage. But if you try it, you’ll see that it only prints one letter to the webpage… the last one. Can you see why? We want to be *adding* each letter to the result variable every time we step through the loop, so we simply have to change two lines:

let userString, upperCaseMinusE, upperCasedString;  
userString = prompt("What do you want to UPPeRCASe?");  
upperCaseMinusE = function(string){  
 let result;  
 result = "";   
 for ( let i = 0; i < string.length ; i = i + 1 ) {  
 let letter;  
 letter = string[i];  
 if ( letter === "e" ) {  
 // Change here.  
 result = result + letter;  
 } else {  
 // And change here.  
 result = result + letter.toUpperCase();  
 }  
 }  
 return result;  
};  
upperCasedString = upperCaseMinusE(userString);  
$("#response").html(upperCasedString);

Save and reload, and you’ll see that it works now just as we would have hoped. If that’s the case, go ahead and commit.

Take a break. We’ve just done a lot. Have a look over the code and make certain you understand what is going on in every line. For the for loop, try writing out the values of result, letter, i, and string for every step through with a made up value for string, like “uppErcase me!”

## Arrays of Objects

I hope you enjoyed your break. Iterating over arrays is a vitally important aspect of programming. In fact, it’s so common that JavaScript has a special method, .forEach(), for iterating over arrays. We could rewrite the function in the previous section this way:

upperCaseMinusE = function(string){  
 let result, stringArray;  
 result = "";   
 // Since forEach() only works on arrays, we have   
 // to convert the string to an array:  
 stringArray = string.split("");  
 // Now we call forEach() on stringArray:  
 stringArray.forEach(function(letter){  
 if ( letter === "e" ) {  
 result = result + letter;  
 } else {  
 result = result + letter.toUpperCase();  
 }  
 }) // Note the parenthesis!  
 return result;  
};

The savings in terms of typing aren’t that great, but .forEach() becomes far more valuable with more complicated arrays.

Let’s imagine that our turtles have cards that tell you about them. On each card, we see the turtle’s name, his favorite color, and his weapon of choice. We can create these cards as JavaScript Objects:

let leonardo, donatello, raphael, michelangelo, turtles;  
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};  
donatello = {name: "Donatello", color: "purple", weapon: "bo"};  
raphael = {name: "Raphael", color: "red", weapon: "sai"};  
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};  
turtles = [leonardo, donatello, raphael, michelangelo];

Each turtle has three properties, .name, .color, and .weapon. And then we put all four turtles into an array, turtles. Now let’s say we want a list of their weapons on the webpage. In scripts.js, type out:

let leonardo, donatello, raphael, michelangelo, turtles, weapons;  
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};  
donatello = {name: "Donatello", color: "purple", weapon: "bo"};  
raphael = {name: "Raphael", color: "red", weapon: "sai"};  
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};  
turtles = [leonardo, donatello, raphael, michelangelo];  
weapons = ""; // a list of weapons.  
$("#response").html(weapons);

Of course, weapons is blank for the time being, so #response on the webpage will just be blank. How can we get the list of weapons, though? We need to iterate over the list of turtles and get each turtle’s individual weapon. Then we can put those together into the weapons string and be on our way. Under weapons = "";, we can add:

turtles.forEach(function(turtle){  
 weapons = weapons + turtle.weapon + " ";  
})

Weapons starts out blank, but then every time through the .forEach() loop, it gets its previous value, plus the value of the turtle’s .weapon property, plus a space. If you save and reload the browser, you should now get a list of all the turtles’ weapons. This is great, but we can do better.

## Mapping and filtering

Let’s say we wanted not only the list of weapons, but we also wanted it in alphabetical order. How could we do that? Well, we know that arrays have the .sort() method, but in order to sort the weapons, we need an array of just the weapons’ names. Currently, turtles is an array of Objects (one for each turtle) and weapons is a string. Instead of .forEach(), we can make use of the .map() method. You use .map() whenever you want to build an array out of another array. It’s a more specific version of .forEach(), but you use it the same way. You write an anonymous function that takes as its first parameter the current array item over which you’re iterating.

let leonardo, donatello, raphael, michelangelo, turtles, weapons;  
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};  
donatello = {name: "Donatello", color: "purple", weapon: "bo"};  
raphael = {name: "Raphael", color: "red", weapon: "sai"};  
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};  
turtles = [leonardo, donatello, raphael, michelangelo];  
weapons = turtles.map(function(turtle){  
 return turtle.weapon;  
});  
// weapons is now ["katana", "bo", "sai", "nunchaku"]  
$("#response").html(weapons);

Now that weapons is an array instead of a string, that means we can also run .sort() on it:

let leonardo, donatello, raphael, michelangelo, turtles, weapons;  
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};  
donatello = {name: "Donatello", color: "purple", weapon: "bo"};  
raphael = {name: "Raphael", color: "red", weapon: "sai"};  
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};  
turtles = [leonardo, donatello, raphael, michelangelo];  
weapons = turtles.map(function(turtle){  
 return turtle.weapon;  
}).sort();  
// weapons is now ["bo", "katana", "nunchaku", "sai"]. Sorted!  
$("#response").html(weapons);

And the webpage is printing “bo,katana,nunchaku,sai”, which isn’t bad, but it looks a bit weird. Let’s replace those commas with commas and spaces:

weapons = turtles.map(function(turtle){  
 return turtle.weapon;  
}).sort().join(", ");  
// weapons is now "bo, katana, nunchaku, sai". Sorted, with commas.  
$("#response").html(weapons);

Notice that weapons is no longer an array. It is now a string. That is because the array’s .join() method creates a string out of an array, where it glues the array’s pieces together using the parameter sent to it, in this case ,. .join() is the opposite of .split(), which turns a string into an array, splitting on the parameter sent to it.

The .map() method opens up possibilities for manipulating data, because as it gives us a new array, we can use other methods inherent to arrays to work on the new data. One such method is .filter().

Say we want a list of the turtles’ names, but only if their names have the letter “o.” Getting the names is easy; it’s no different than getting the weapons:

let names;  
names = turtles.map(function(turtle){  
 return turtle.name;  
}).sort().join(", ");  
$("#response").html(names);

This gets us most of the way there, but Raphael is in the list, and we want him gone. .filter() works just like .map(), but instead of returning a value, it returns the value over which it is iterating only if it meets a conditional, like an if statement. Now, strings have a method, .includes(), that returns true if the string includes whatever the parameter is. Let’s add some code, then.

let names, namesWithO;  
names = turtles.map(function(turtle){  
 return turtle.name;  
}).sort();  
namesWithO = names.filter(function(name){  
 return name.includes("o");  
}).join(", ");  
$("#response").html(namesWithO);

Because "Leonardo".includes("o") returns true, that name is included in the list. Because "Raphael".includes("o") returns false, it is not included.

## Exercises

1. Write a function that takes an array of integers and, using .map(), returns an array of those integers, doubled. So if we give it [1, 2, 3], we receive, in turn, [2, 4, 6].
2. Add functionality to the weapons examples above so that the final result is “bo, katana, nunchaku, and sai.”