Collections of Data

“Data,” you may recall, is a plural. Just like “bacteria” is a collection of many instances of a single “bacterium,” so it is with data. Data are a collection of single “datums.” So far, we’ve been working for the most part with just single pieces of information, like a single number. We’ve been using that number with other numbers, of course, but not as a collection of information.

A collection of information combines to form something larger than it itself. Perhaps the easiest collection to think of in programming terms is a list, like, say, a to-do list. The items on the list don’t interact with each other, necessarily (what does “schedule dentist appointment” have to do with “buy kitty litter”?), but, in a list, they can be ordered and shuffled around. Furthermore, by being in a list, each chore is exposed to **iterability**, the ability to loop over them.

Iterability it a crucial concept in programming. We’ve already seen it in action with while and for loops, but with collections of data, it becomes even more powerful. Let’s take another list, a list of friends. It’s your birthday, and you want to invite them to a party. But you also want the invitations to be “personalized.” You could iterate over (loop over) your list, get your friends’ first names, and then use that name in an email that opens “Dear FIRST\_NAME.” They each get the same email, but the first name matches their own.

## Arrays

In JavaScript, the simplest list data structure is called an **array**. Arrays are common in programming languages, and they are typically designed to be extremely fast at sorting and iterating. In JavaScript, we’re not so lucky; arrays don’t offer the same kind of speed benefits. Still, they are crucial, and we’ll be seeing them a lot from now on.

Simply put, an array is a set of pieces of data surrounded by brackets ([]). The following are all valid arrays, and type them into your scripts.js (you can delete everything from the previous chapter):

let arrayOfStrings, arrayOfNumbers, arrayMixed;  
arrayOfStrings = ["a", "b", "c"];  
arrayOfNumbers = [1, 2, 3];  
arrayMixed = ["a", 1, null, true, arrayOfNumbers, [4.5, 5.6]];

Notice that you are not limited to a single data type in an array. Strings, numbers, null, true, variables, and even other arrays can be used as the contents of arrays. Later on, when we start building maps, we will have at least one array made up of geographical points.

Each item in an array can be accessed by its **index**, which is an integer unique to that item. The indices begin with 0, which is confusing for beginners. So if you add to the above:

$("#response").html(arrayOfStrings[2]);

#response will read “c.” The third value of the array is "c", but its index is 2, because the index begins with 0. So to get “a,” we would call $("#response").html(arrayOfStrings[0]);. We are interested in the zeroth value. Again, I know this is confusing, but you will get the hang of it with practice, and then you can join that exclusive club of people who make jokes about zero-based numbering.

## Objects

Above, when I mentioned that arrays are a bit peculiar in JavaScript, that is because arrays are a simplified version of the JavaScript **object**. Because “object” is such a common word in English, in this text, from now on, when you see Object, you know that I mean, specifically, this generic JavaScript data type. Where arrays are pieces of data surrounded by brackets, Objects are surrounded by braces ({}). Similarly, while arrays have indices, Objects have **properties**. Let’s define an Object.

let myBurritoObject;  
myBurritoObject = {  
 tortilla: "wheat",  
 guacamole: true,  
 beans: "pinto",  
 habaneroSauceSquirts: 3  
 };  
// and let’s access a property  
$("#response").html(myBurritoObject["tortilla"]);

With an array, we call it using the syntax arrayName[indexNumber]. With an Object, we replace the index with a property. But we can do even better:

$("#response").html(myBurritoObject.tortilla);

It’s much more succinct to use **dot-notation** to access properties.[[1]](#footnote-23) In fact, for the rest of this text, whenever I refer to a property, I’ll refer to it as a .property. Properties are especially useful because arrays have them as well. For example, every array has a .length property:

let arrayOfStrings;  
arrayOfStrings = ["a", "b", "c"];  
$("#response").html(arrayOfStrings.length);

This will print “3,” because the value of that array’s .length property, or its length, is three. So even though the largest *index* value in the array is 2, its length is 3.

Objects can contain other Objects, of course, but we really start cooking when we build arrays of Objects. Those points on a map I mentioned before? They will be an array of Objects, where each Object has properties that give its place name and its coordinates.

## Methods

JavaScript, like Ruby, is famous because in both languages, *everything* is an Object. Objects are Objects, arrays are Objects, strings are Objects (in that they have properties, as we’ll see below), null is an Object, and even functions are Objects. Since a property of an Object can be any other kind of Object, that means that a property can even be a function. For example, to return to myBurritoObject, you can add a new property:

let myHabaneroSauceSquirts, myBurritoObject;  
// First, define and assign a variable for how   
// spicy the burrito is.  
myHabaneroSauceSquirts = 3;  
// Now assign the burrito object.  
myBurritoObject = {  
 tortilla: "wheat",  
 guacamole: true,  
 beans: "pinto",  
 // Make use of the variable above.  
 habaneroSauceSquirts: myHabaneroSauceSquirts,  
 // Use the variable again in a function.  
 spiciness: function(){  
 if (myHabaneroSauceSquirts > 0 ){  
 alert("This is a spicy burrito!");  
 } else {  
 alert("This is a mild burrito.");  
 }  
 }  
};  
$("#response").html("Your burrito has " +  
 myBurritoObject.habaneroSauceSquirts +  
 " squirts of habanero.");  
myBurritoObject.spiciness();

Save, and reload, and see what happens. If you’re told the burrito is spicy, commit. Now let’s have a look at the two new things I’m presenting here. The property here, .spiciness is actually a function, and it is defined in a way similar to how we have been defining functions all along. That is, all along we have been writing **anonymous functions**. They are anonymous in that they are ephemeral. They exist and then they’re gone. When we create a function like:

let makeABurrito = function(){  
 // Do stuff.  
};

The function(){} part of it disappears into the variable makeABurrito. We can then resurrect it using makeABurrito(). In .spiciness, however, we are assigning the function to a *property*, not even a variable. Later, we will make even more ephemeral anonymous functions, where the function gets called, executed, and then disappears, without even a variable or property to resurrect it. But anonymous functions pop up all over the place in JavaScript, which is why I promised last chapter that you would be typing function a lot.

When properties are functions, they are called **methods**. Methods are built into the Object. To use an example we’ve already seen, every console Object has the .log() method built in. Arrays also have a series of useful methods:

let turtles, sortedTurtles, reversedTurtles, turtleNames;  
turtles = ["Leonardo", "Donatello", "Raphael", "Michelangelo"];  
sortedTurtles = turtles.sort();  
// sortedTurtles is:  
// ["Donatello", "Leonardo", "Michelangelo", "Raphael"]  
reversedTurtles = turtles.reverse();  
// reversedTurtles is:   
// ["Raphael", "Michelangelo", "Donatello", "Leonardo"]  
turtleNames = turtles.join(" ");  
// turtleNames is "Leonardo Donatello Raphael Michelangelo"  
turtles.push("Splinter");  
// turtles is now:   
// ["Leonardo", "Donatello", "Raphael", "Michelangelo", "Splinter"]  
turtles.pop();  
// back to ["Leonardo", "Donatello", "Raphael", "Michelangelo"]

Note that .sort(), .reverse(), and .join() do not change the value of turtles. Instead, we define new variables, sortedTurtles, reversedTurtles, and turtleNames. Then we assign to those variables two new arrays and a string. .pop() and .push(), however, *do* change turtles.

## Strings as arraylike things

Because everything is an Object, that includes strings. Strings can behave a bit like arrays, but they also, as Objects, have properties and methods. I’ll mention a few here, because manipulating strings (or “text”) is a vital feature of writing web pages.

let string, firstLetter, stringLength;  
string = "This is a string.";  
// Strings have indices and lengths, just like arrays:  
firstLetter = string[0];  
// firstLetter is "T"  
stringLength = string.length;  
// stringLength is 17  
//  
// Strings also have methods, just like arrays:  
let upperCaseString, replacedString;  
upperCaseString = string.toUpperCase();  
// upperCaseString is "THIS IS A STRING."  
replacedString = string.replace("string", "pipe");  
// replacedString is "This is a pipe."

## Exercises

1. Write a function that always returns the last item in whatever array you pass it.
2. Numbers also have methods and properties. Look them up at [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) and change your webpage so that it asks for a number and tells you if it is an integer or not.

## Footnotes

1. Dot-notation does not work, however, for index values. arrayOfStrings.1 will cause an error. [↑](#footnote-ref-23)