Functions and Objects

Christopher M. Harden

Contents

1 Arrays

It is often convenient to be able to store multiple pieces of data in a single variable. An example of this would be a list of ingredients. While it would be possible to create multiple variables such as **var** firstIngredient , **var** secondIngredient, **var** thirdIngredient, and so on, this is not easy to work with inside a script. A better way would be to have an ingredients list, and then refer to the first, second, and third ingredient in the list.

1.1 Defining arrays

An array is how we store lists in JavaScript. Arrays are declared with square brackets. Inside the square brackets goes your list of items, separated by commas. You can store any type of variable inside an array, and each item in an array can have a different type. However, unlike other types, JavaScript will not automatically turn the previously mentioned types into an array. Thus you will sometimes see empty arrays being assigned to variables, such as **var** ingredients = [], if the list of ingredients is not known in advance.

Once you have an array you use square brackets to retrieve the data inside them. This process is called indexing. Arrays have integer indexes starting from 0. Thus the first element has index 0, the second element has index 1, and so on. For example, to get the second ingredient in a list (stored in an array) you would write ingredients [1] .

Note. *The fact that humans count from 1 and computers count from 0 is an endless source of bugs and errors. The reason for computers counting from 0 is historical and comes from earlier, lower-level programming languages such as C. For example, in C, memory is accessed via its address, which is an integer representing blocks of memory. An array would simply be a continuous block of memory, and you would be given the address of the first block. Thus to access the first element, you would look up that address. To access the second element, you would add 1 to the address and then loop it up. To access the third element, you would add 2 before looking up, and so on. As you can see, to get element n you would add $n - 1$ to the address, and from this it is easy to see why counting from 0 makes sense.*

JavaScript is not bound by the same restrictions as C, but counts from 0 anyway as tradition. Simply put, it is a convention you will get used to over time.

Once you have indexed the array you have a regular variable just like in previous classes. Just like normal you can assign to it or use it in various calculations. If you happen to index an element that doesn't exist, JavaScript

returns **undefined** since no data was stored there. Listing 1 gives a quick overview of this as a small script.

Listing 1: Using arrays

```
var emptyArray = [];
var mixedArray = ["Peter", 17, "Glasgow"];

var daysOfWeek = [
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday"
];

alert("Today is " + daysOfWeek[2]); // Tuesday

// Make Peter older
mixedArray[1] = 18;

// Demonstrating that you can store whatever you like
// inside an array
console.log(typeof mixedArray[0]); // "string"
console.log(typeof mixedArray[1]); // "number"
console.log(typeof mixedArray[2]); // "string"

// Shows you the entire array.
// Format depends on web browser
console.log(mixedArray);
```

1.2 Array length

Looping constructs are often seen whenever arrays are used. Arrays are generally used to store related data, and so there is often a need to do some particular action on every element of an array. However, to loop over an array we need to know what indexes there are in the array, and we can only know that if we know its size. In most cases we cannot assume the array's size. Fortunately, JavaScript gives us a way to simply ask the array itself,

since it knows how many elements it has inside it. If arr is an array then arr.length will tell us the size of arr. More specifically, it returns the smallest integer n such that arr[m] is **undefined** for all $m \geq n$.

Listing 2 has a named block called makeArray, which creates an array of random size representing a list of people and randomly decides whether that person is male or not. This named block is supposed to represent some code somewhere else in a script. Consequently, the size of the array is lost after the makeArray block ends due to scoping. However, in the useArray block, we can still loop over the array since we can directly ask the array for its own size. We use this to count the number of males in the list.

Listing 2: Iterating over an array

```
1  var isMale = [];
2
3  makeArray: {
4    const numberOfPeople = Math.floor(100*Math.random()
       + 1);
5    for (let i = 0; i < numberOfPeople; i++) {
6      isMale[i] = (Math.random() < 0.5);
7    }
8  }
9
10 useArray : {
11   var numberOfMales = 0;
12
13   for (let i = 0; i < isMale.length; i++) {
14     if (isMale[i]) numberOfMales++;
15   }
16
17   console.log("There are " + numberOfMales + "
       males.");
18 }
```

Since JavaScript allows you to store anything inside an array, that includes **undefined**. This means that you can skip indexes if you so choose. As mentioned before, these undefined indexes have value **undefined**. However the words size and length are no longer interchangeable. Listing 3 creates two arrays with length 5, but one of the arrays only has 4 items in it. This is why length was defined above in terms of the smallest number with a given property, since the naïve definition based on the number of elements (i.e. size) can be misleading.

Note. *It goes without saying that because of this you should not skip indexes without a very good reason.*

Listing 3: Size vs Length

```
var arr = [1, 2, 3, 4, 5];

// Demonstrate that this array has 5 elements
console.log(arr);
console.log(arr.length);

// Redefining arr. Notice how I skip arr[2]
arr = [];
arr[0] = 1;
arr[1] = 2;
arr[3] = 3;
arr[4] = 4;

// Now arr has 4 elements, but it is still of length 5
console.log(arr);
console.log(arr.length);
```

1.3 Multi-dimensional arrays

As mentioned a few times previously, JavaScript arrays can store anything. In particular, an array can store other arrays. This is called a multi-dimensional array, and is often used for storing tabular data. If arr [0] contains an array, then you can immediately index again to get the contents of that array. For example, arr [0][2] will grab the first row of arr, and then get the third column of that row. Listing 4 uses this to create a three-dimensional array. This array represents the seating arrangements at a wedding. There are three tables, each table has two sides, and each side has three chairs. White space is added to the array to line up the opening and closing brackets, to make sure we don't accidentally miss one.

Listing 4: A three-dimensional array

```
var seating = [
  [
    [
      ["Peter", "David", "Mark"],
      ["May", "Helen", "Jill"]
```

```
      ]
    ],
    [
      [
        ["Natasha", "Roxanne", "Lana"],
        ["Scott", "Emmet", "Darren"]
      ]
    ],
    [
      [
        ["Francis", "Christina", "Cedric"],
        ["Nicole", "Albert", "Shelia"]
      ]
    ]
];

// Who is sitting on the third table, in the middle of
   the right side?
console.log(seating[2][1][1])
```

2 Functions

In previous exercises we have seen things such as alert(), console.log(), Math.random() and so on. In the case of Math.random(), it gives us random numbers between 0 and 1, including 0 and excluding 1. While we could in theory look up the mathematical theory of Mersenne Twisters and re-implement that algorithm ourselves in every single script we write, it is not advised. Generating random numbers is a surprisingly difficult task, and mistakes may not be spotted over thousands of runs of our scripts. The far superior option is to have somebody with experience with mathematics write this algorithm for us, and we simply call it as needed. Our first step into such code reuse is the idea of a function.

2.1 Defining and calling functions

Functions come in two parts: the definition and the calling. Listing 5 shows a function definition. Line 1 starts with the **function** keyword, telling the JavaScript interpreter that a function is about to be defined. Next comes the name of the function, which you may define just like every other identifier.

Next comes a matching pair of parentheses, which becomes important later. Finally there is a block containing some code.

Listing 5: Defining a function

```
function hello() {
  alert("Hello everyone!");
}
```

Whenever you wish to run the function, you call it with `hello();`. You can call it as many times as you wish, and the function will run every time.

2.2 Parameters and return values

Listing 5 is a simple function that doesn't do very much. It has no data to work with, so it does the same thing over and over. Data can be passed into a function via its parameters. You may take as many parameters as you want. To take parameters, you add a comma separated list of variable names between the set of parentheses mentioned before. Inside your function you may use these variables just like any other variables you have used before. You may also define your own variables inside a function. Any variables you define inside a function are scoped to that function. Thus you will not accidentally overwrite any variables outside the function.

Note. *Consider the inside of a function as a 'clean space'. You may do whatever you like inside the function, without affecting the outside world. There are various tricks to break the confines of a function and affect the outside world, but you should avoid them. There is almost never a good reason to do this, and it makes it difficult to know what your function will do if you do. Breaking this promise tends to result in 'actions at a distance', where (for example) your image gallery happens to break randomly because an analytic script you use that was written by another company in another language in another country just happens to have picked the same variable name.*

Functions may also return a single value. For example, Math.random() should give you the random number it generated. To do that we have the **return** keyword, which immediately ends the function. If anything follows **return** it becomes the return value of the function. If the **return** keyword is by itself, the function returns nothing.

Note. *Of course, using arrays (and objects, which are mentioned later) as a return value means that your single return value actually consists of multiple pieces of data. So there is never a need to return multiple values.*

When you call a function, you may fill in parameters between the parenthesis, and use the function to get a return value. If you do not fill in all the parameters the rest automatically default to **undefined**. Similarly, if the function does not return a value then you are given **undefined**.

Note. *You can define a function to default to a different value. Unfortunately this is not common across all web browsers. If you wish to take optional parameters, you will need to define a default inside your function using code similar to the following:*

```
var data = (param === undefined) ? default : param
```

Listing 6 uses the above to calculate the area of 5 circles of random radius. A function is defined that deals with finding the area of a circle, using its own copy of π to 50 decimal places just in case another function is using a different approximation.

Listing 6: Defining a function

```
function areaOfCircle(radius) {
  const pi =
     3.14159265358979323846264338327950288419716939937510;
  return pi * radius * radius;
}

for (let i = 0; i < 5; i++) {
  let r = Math.floor(100*Math.random() + 1);
  let a = areaOfCircle(r);

  console.log(
      "Radius: " + b
    + ", Area: " + a
  );
}
```

3 Objects

Arrays are useful for storing data that has some natural order to it. Objects can be considered to be a more general version of an array, which is appropriate for representing more complicated data. Objects associate a string, called a key, with a value. While all keys are strings the values associated

with these keys can be any JavaScript type. In particular this includes arrays and other objects.

3.1 Creating and using objects

Objects can be initialized with curly brackets. Inside these brackets are a comma separated list of key-value pairs separated by a colon. Since keys are automatically strings, you do not need to quote them if they do not contain symbols such as spaces.

Objects are indexed just like arrays. However, if the key is known in advance and it satisfies the requirements to be a JavaScript identifier, you may use dot notation instead. Dot notation is preferred when available. Just like with arrays non-existent keys will return **undefined**. If you have no more use for a key, delete obj.key will delete key and its value from obj.

Listing 7 represents three people as objects and shuffles their relationships around.

Listing 7: Representing people as objects

```
var stephen = {
  name : "Stephen",
  age  : 23
};
var jessica = {
  name : "Jessica",
  age  : 21
};
var samantha = {
  name : "Samantha",
  age  : 28
};

console.log(stephen["name"]);
console.log(stephen.age);

// Create a two way relationship
stephen.likes = jessica;
jessica.likes = stephen;
// And a one way relationship
samantha.likes = jessica;

// And now we make this love story more complicated
```

```
24 delete stephen.likes;
25 samantha.likes = stephen;
26
27 console.log(jessica.name + " likes " +
       jessica.likes.name);
28 console.log(samantha.name + " likes " +
       samantha.likes.name);
```

3.2 The in keyword

Since keys can be added and removed from an object at will, we need a way to check whether or not a key exists in a given object. The **in** keyword is used to test for a key's existence. It takes the form key **in** obj. Since keys are strings you almost always want to quote key in double quotes, unless you actually want to look for the string inside the variable key as the test key.

Note. *Since undefined keys return* ***undefined*** *when accessed, you might think that testing for* ***undefined*** *will tell you if a key exists or not. However this is not true. There is nothing stopping you from declaring a key with a value of* ***undefined****. This key exists, but its value does not. However without using* ***in*** *you cannot tell the difference!*

The **in** keyword also serves a second purpose inside a for loop. It can be used to get all the keys of an object. This allows you to check the contents of an object. Listing 8 uses objects inside objects to represent people attending a party, and whether or not they are vegetarian or suffer from a nut allergy. We count the number of vegetarians attending so the chefs can prepare in advance. We also emit a warning if anyone has listed themselves as having a nut allergy, since even accidental contamination can be fatal.

Listing 8: Iterating over an object

```
1  var foodPrefs = {
2    Stacy   : { veggie : true,  nut : false },
3    Nicole  : { veggie : false, nut : false },
4    David   : { veggie : false, nut : true },
5    Daniel  : { veggie : true,  nut : false }
6  };
7
8  let noOfVeg = 0;
9  let nutAllergies = false;
10
```

```
11 for (let i in foodPrefs) {
12   if (foodPrefs[i].veggie) {
13     noOfVeg++;
14   }
15   if (foodPrefs[i].nut) {
16     nutAllergies = true;
17   }
18 }
19
20 if (noOfVeg === 0) {
21   alert("No vegetarians are attending this party.");
22 } else {
23   alert("At least " + noOfVeg + " vegetarian meals are
       required.");
24 }
25
26 if (nutAllergies) {
27   alert("WARNING: chefs must be informed of potential
       nut allergy.");
28 }
```

Note. *While this sounds like an appealing way to iterate over an array, it should not be used for that purpose. The keys are not returned in any particular order, so this will not in general count from 0, 1, 2, 3, etc. Additionally it is possible for an array to have keys other than its indexes, and the* ***in*** *will pick those up without you expecting them. Finally, since all keys are strings, attempting to add with them will cause string concatenation instead due to type coercion.*

4 The function/object relationship

4.1 First-class objects

Functions are not just something that can be called in JavaScript, but rather something that can be manipulated just like every other type of variable you have seen so far. In JavaScript, we say that functions are first-class objects. Here the word 'objects' refers to JavaScript objects. Indeed, every function you have defined so far has been converted into a **Function** object stored inside a variable with the same name as the function's name. The parenthesis are considered an operator that works on **Function** objects. This piece of

technical detail has some interesting applications. One such application is that functions can be passed inside other functions. This is the basis of an entire programming style called functional programming, which works with objects called higher-order functions.

Since functions are passed around as objects, it is convenient to be able to create functions immediately when they are required. We can do this by creating an anonymous function. An anonymous function has no name, and is defined with the **function** keyword followed by an pair of parenthesis defining arguments. To use such a function, you must either assign it to a variable or immediately call it with another pair of parenthesis. Since the function has no name, there is no way to call the function directly.

One of the easiest higher-order functions to understand is the map function. It takes an array and a function as a parameter, and returns an array. For every element in the input array it applies the function to that element and assigns the result to the output array. Listing 9 defines map in JavaScript. Line 17 passes the sayHello() function into map. Lines 18-20 do a similar thing to say goodbye to everyone, except it anonymously defines the function right at the point it is used.

Listing 9: Implementing map

```
1  function map(arr, func) {
2    var results = [];
3
4    for (let i = 0; i < arr.length; i++) {
5      results[i] = func(arr[i]);
6    }
7
8    return results;
9  }
10
11 function sayHello(name) {
12   console.log("Hello " + name);
13 }
14
15 var names = ["Peter", "David", "Susan"];
16
17 map(names, sayHello);
18 map(names, function(name){
19   console.log("Goodbye " + name);
20 });
```

Note. *In more modern browsers the Array object has some built-in higher-order functions as methods, including the map function just discussed. However, map does not exist in older browsers and other higher-order functions such as foldl and foldr do not exist at all. In actual code you should use* ***if*** *statements to check if you have a map method and use it if available, and only re-implement it if it doesn't exist. This process is called polyfilling, and many JavaScript libraries exist which are devoted just to polyfilling.*

4.2 Closures

Since functions are objects, and objects can be returned from a function, it follows that functions can also be returned from a function. Functions have access to variables within their scope as discussed in previous classes. However because functions can be passed around in variables they can end up in a completely different scope where their variables no longer exist. Despite this functions can continue to use their variables, as the JavaScript interpreter makes sure that each function has access to everything it should. The process which makes this work is called a closure.

Listing 10 implements the sayHello() and sayGoodbye() functions referred to in listing 9. This is done via a greetName() function that creates these functions. Since greeting is a parameter to greetName(), and the function returns another function, the function returned should have no access to greeting any more and thus should cease to work. However, via the power of closures these returned functions keep their reference to the greeting that existed when it was created.

Listing 10: Creating a closure

```
1  function greetName(greeting) {
2    return function(name) {
3      console.log(greeting + " " + name);
4    }
5  }
6
7  sayHello = greetName("Hello");
8  sayGoodbye = greetName("Goodbye");
9
10 // Demonstrate that these are the required functions
11 sayHello("Nicole");
12 sayGoodbye("Nicole");
13
```

```
14 // All of these produce errors. You have no access to
      either greeting
15 greeting;
16 sayHello.greeting;
17 sayGoodbye.greeting;
```

4.3 Constructors

When we use curly brackets to define an object, what we are technically doing is creating an Object object and assigning some keys and values to it. Since objects tend to represent something, we can name our type of object. Not only is this useful to trace errors the console shows us, but it is also useful to ensure that every object is constructed correctly. When working with objects nested inside objects it is especially easy to accidentally mistype after creating multiple copies of an object, or to accidentally create an inconsistent object. For example, if you are representing people as objects, you don't want to declare them as simultaneously dead and alive.

To help with this we can define a function called a constructor. A constructor creates an object on our behalf. Since a constructor is a function, it can take whatever parameters you need to help you create the object. Constructors are typically given names starting with a capital letter, but this is merely convention. Since the constructor doesn't know what object it is creating, the **this** keyword is introduced to represent the object it is working on. Once a constructor is defined we use the **new** keyword to call the constructor. Listing 11 creates a constructor to represent the object used in listing 8 to represent food preferences and allergies. We also use the constructor to perform some basic error checking, in this case ensuring that the Preferences object only contains booleans.

Listing 11: Using constructors

```
1  function Preferences(veggie, nut) {
2    this.veggie = !!veggie;
3    this.nut    = !!nut;
4  }
5
6  var foodPrefs = {
7    Stacy   : new Preferences(true, false),
8    Nicole  : new Preferences(false, false),
9    David   : new Preferences(false, true),
10   Daniel  : new Preferences(true, false)
```

```
11 };
```

4.4 Methods

Particularly complicated objects generally require their own functions to handle them. Objects can be given keys with function values to store these functions. These functions are called methods. For example, console is an object that represents a developer's console inside a web browser. Logging to it is a function that makes sense only for a console, and as such there is a console.log() method on the console object. Your objects can also have methods associated with them, and assigning these methods is generally left to the constructor. Listing 12 creates an Person object with a name, and two functions which get and set that person's age.

Listing 12: Adding methods to your objects

```
1  function Person(name) {
2    this.name = name;
3
4    var age = null;
5
6    this.getAge = function() {
7      return this.age;
8    };
9
10   this.setAge = function(age) {
11     if (age === null) {
12       this.age = null;
13     } else {
14       age = Number(age);
15       this.age = Number.isNaN(age) ? null : age;
16     }
17   }
18 }
```

Line 4 creates a variable inside the Person constructor. This variable would fall out of scope after the function ends, but because JavaScript has closures and because there are two methods, getAge() and setAge() that access that variable, the variable stays accessible but only by a Person object. Thus all access to the age variable passes through the Person methods. There are two main benefits of this:

1. A Person object can assume that it has a valid age, since the only function that sets age is the setAge() method, and that method ensures that only valid ages are set.

2. A Person object can be changed later to store a date of birth instead of an age, and as long as you re-implement getAge() and setAge() correctly no other changes to your script will be needed, since no-one else could possibly touch age directly.

When designing your own objects, you should think carefully about what keys and methods should be accessible to the outside world, and only expose a simple interface between you and your object. Similarly, when using objects designed by others, you should stick to the interface you are given and should not attempt to break into the internals of the object. Together, functions and objects work together to split scripts into small, manageable chunks, which are easier to test for bugs and other errors. Well designed functions and objects can even be used on multiple websites without any modification whatsoever.

4.5 Constant objects

You can declare constants with the **const** keyword mentioned in the first class. This keyword ensures that a variable is not reassigned. However objects can have keys added and removed at will, and so declaring them as constant will not do as you would expect. While the object cannot be reassigned for a new object, we can manipulate the old object into looking like the new object, which defeats the purpose of a constant. To stop this tampering of your objects there are two levels of 'constantness' you can define. In strict mode attempting to break this protection will result in an immediate error, while non-strict mode will just silently ignore your attempts.

- Object.seal() takes an object and stops any addition or deletion of keys. Keys that exist when the object is sealed can still be modified at will.

- Object.freeze() takes an object and stops all further changes. Any change of any kind results in either an error or silent ignoring.

Listing 13 defines a constant object representing a person, which we immediately manipulate into looking like an item on a supermarket shelf. We use Object.seal() to stop any further changes to the keys, and show that we can still change the name of the item since name existed when the seal

was applied. Finally we freeze the object before we get tricked into selling a luxury mansion for £5.50.

Listing 13: Sealing and freezing objects

```
const mia = {
  name : "Mia",
  age  : 17
};

// This will fail since we cannot reassign
mia = {
  name : "Chicken",
  priceGBP : 5.50
};

// We can just bypass this check by mutating a person
   into a food item
mia.name = "Chicken";
delete mia.age;
mia.priceGBP = 5.50;

// This will stop us from changing the keys, but we
   can still edit the contents
Object.seal(mia);
mia.priceUSD = 8.30; // An error
mia.name = "Can of baked beans"; // Fine!

// And now the object is truly constant.
Object.freeze(mia);
mia.name = "Deed to a luxury mansion"; // An error
```

5 Try it yourself

To check this week's knowledge, try the following tasks below. Each task should take no more than a few minutes, and does not require information from a future class.

1. Consider a web site that sells music online.

   (a) Design an object that represents a song or music track, and create the appropriate constructor.

(b) Design an object that represents an album, and create the appropriate constructor. Give your album object a method that adds a given song to the end of the album.

(c) Assuming your song objects have a property which represents a rating, write two functions that return the best and worst songs on any given album.

2. There are two expressions which returns a random integer depending on the values of min and max. These are:

```
1 Math.round(Math.random()*(max - min + 1) + min)
2 Math.floor(Math.random()*(max - min + 1) + min)
```

(a) Create two functions for each expression.

(b) Generate 1,000,000 random integers between 1 and 10 using each function. For each function count how many times each function generates a particular integer.

(c) Determine if either expression ever generates a random number outside of its supposed range.

(d) Looking at your two sets of results, determine which expression (if any) fairly generates integers.

(e) If you were given the task of randomly choosing images to display in an image gallery, which expression (if any) would you use?