{ Array Basics. }

Objectives

By the end of this chapter, you should be able to:

- Create a JavaScript array
- Access elements in an array
- Modify the values stored in an array
- Use `push` and `unshift` to add elements to an array
- Use `pop` and `shift` to remove elements from an array
- Use `splice` for more advanced array modifications

Introduction

So far, we've seen five different primitive data types in JavaScript: `string`, `number`, `boolean`, `null`, and `undefined`. We've also seen how to store these values inside of variables.

Sometimes, however, you need a more complex data structure when building your application. For example, maybe you need a list of restaurant names so that you can display each one to a user when she's looking for a place to eat nearby. Or maybe you're doing some math and want to maintain a list of prime numbers. It would be pretty annoying to have to write

```
let firstPrime = 2;
let secondPrime = 3;
let thirdPrime = 5;
let fourthPrime = 7;
let fifthPrime = 11;
```

This is fine if you know how many primes you need at the outset, but what if you didn't know how many values you needed to store? Or what if you did know how many values

you needed, but the number was quite large? Writing out a variable for each one can quickly become unmanageable.

Thankfully, JavaScript provides you with a data type to help in these situations: the *array*. You can think of an array as simply a list of values.

To write an array in JavaScript, you use square brackets `[]` and comma separate each value in the array. Here are some examples:

```
let primes = [2, 3, 5, 7, 11];
let names = ["Alice", "Bob", "Charlie"];
let booleans = [true, false, false, true];
let mixedTypes = [1, "sweet", true, null, NaN, "bye!"];
let woahhh = ["What's up with this? -->", ["Woah", "crazy!"]];
let emptyArray = [];
```

You can put anything you want inside of an array: numbers (as in `primes`), strings (as in `names`), booleans (as in `booleans`), and other primitive types are all fair game. You can also have multiple different types inside of an array: just because the first element in an array is a number doesn't mean that every subsequent element needs to be a number too. For example, `mixedTypes` has many different types inside of it. You can even store arrays inside of other arrays, as in the `woahhh` array above!

At this point, you may be wondering why we didn't mention arrays when we talked about other data types in JavaScript. The reason is that up until now, we've been dealing with *primitive* data types in JavaScript. But arrays aren't primitives; they're examples of what's called a reference type. We'll talk about reference types in more detail in the next chapter. For now, it's sufficient to note that

```
typeof [1, 2, 3]
```

returns `object`. So arrays are a type of object, which we'll talk about in more general terms later.

Accessing and updating array values

To access an element in an array, we specify the name of the array followed by sqare brackets brackets and the *position* (also called the *index*) of the element we're trying to access. **Arrays are zero-indexed**, which means that the first element is accessed at index 0. Let's look at an example:

```
let arr = [5,3,10];
arr[0]; // should equal 5
arr[1]; // should equal 3
arr[2]; // should equal 10
arr[3]; // should be undefined -- remember, arrays are zero-indexed!
arr[1+1]; // the same as arr[2], which is 10
arr[arr.length-1]; // shorthand for the last element of an array, in this case 10
```

To update a value in an array, we can simply assign an element at a given index to a new value:

```
let arr = [5, 3, 10];
arr[0] = -1000;
arr[2] = "dope";
arr; // should be [-1000, 3, "dope"]
```

Adding to arrays

There are a number of ways you can add elements to an array.

One way is by setting a value at a new index in the array.

```
let arr = [1,2,3];
arr[3] = 4;
arr; // [1,2,3,4]
```

Be careful with this approach, though -- you can add an element at any index, and any elements that don't have values in them will be filled with `undefined` values.

```
let arr = [1,2,3];
arr[5] = "whoa";
arr; // [1, 2, 3, undefined, undefined, "woah"]
```

If you want to add to the end of an array, a better approach is to use the `push` function - this function returns the new length (the number of elements) of the array.

```
let arr = [3, 2, 5];
arr.push(7); // returns the new length, i.e. 4
arr; // [3, 2, 5, 7]
```

On the other hand, if you want to add to the beginning of an array, you can use the `unshift` function. As with `push`, `unshift` returns the length of the modified array.

```
let arr = [1,2,3];
arr.unshift(0); // returns the new length, i.e. 4
arr; // [0,1,2,3]
```

Removing from arrays

We've seen how we can add elements from arrays. But what about removing elements?

One (not common) way to remove elements is to manually set the length of the array to a number smaller than its current length. For example:

```
let arr = [1,2,3];
arr.length = 2; // returns the new length
arr; // [1,2]
```

A more common way to remove elements from the back of an array is to use `pop()`. This function works in sort of the opposite way as `push`, by removing items one by one from the back of the array. Unlike `push`, however, `pop` doesn't return the length of the new array; instead, it returns the value that was just removed.

```
let arr = [1,2,3];
arr.pop(); // returns 3
arr; // [1,2]
```

If you want to remove an element from the *front* of an array, you should `shift()` (like `unshift`, but the opposite)! As with `pop()`, `shift()` returns the removed value.

```
let arr = [1,2,3];
arr.shift(); // returns 1
arr; // [2,3]
```

There's also a `delete` keyword in JavaScript, which you might think could be used to delete elements in an array. However, this isn't quite how `delete` works with arrays. When you use this keyword, the value at the index where you delete will simply be replaced by `undefined`. This usually isn't what you want, which is why you won't often see people use `delete` on arrays. It's more common to see this word used with objects, which we'll talk more about in the next unit.

```
let arr = [5, 4, 3, 2];
delete arr[1];
arr; // [5, undefined, 3, 2]
```

Removing/Adding or both with splice

One of the more powerful array methods is `splice`, which allows you to either add to an array or remove elements or even do both! You can think of `splice` as a powerful generalization of `push`, `pop`, `unshift`, and `shift` all in one!

The splice method accepts at least two arguments. The first argument is the starting index, indicating where values will be removed or added. The second parameter is the number of values to remove. Optionally, you can pass in an unlimited number of additional arguments; these correspond to values you'd like to add to the array. The splice method always returns an array of the **removed** elements. Here are some examples:

```
let arr = [1,2,3,4];
arr.splice(0,1); // returns [1]
arr; // [2,3,4]

let arr = [1,2,3,4];
```

```
arr.splice(0,1,5); // returns [1]
arr; // [5,2,3,4]

let arr = ["a","b","c","d"];
arr.splice(1,2,"x","y","z"); // ["b", "c"]
arr; // ["a", "x", "y", "z", "d"]
```

Exercises

1. Create an array of your favorite foods (call it `favoriteFoods`). Make sure it has at least three elements.
2. Access the second element in `favoriteFoods`.
3. Change the last element in `favoriteFoods` to some other food.
4. Remove the first element in `favoriteFoods` and store it in a variable called `formerFavoriteFood`.
5. Add a favorite food to the back of the `favoriteFoods` array.
6. Add a favorite food to the front of the `favoriteFoods` array.
7. What happens when you try to `pop` from an empty array?
8. In the examples below, use `splice` to convert the first array to the second array:
    - `[2, 3, 4, 5]` -> `[2, 4, 5]`
    - `["alpha", "gamma", "delta"]` -> `["alpha", "beta", "gamma", "delta"]`
    - `[10,-10,-5,-3,2,1]` -> `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`