Arrays

(https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-arrays) (https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-arrays/issues/new)

Learning Goals

- Identify data structures and `Array`s
- Create `Array`s
- Access the elements in an `Array`
- Learn about nested `Array`s
- Introduce `Array` methods
- Understand mutability

Introduction

So far, we've been focusing on JavaScript's *primitive* data types — types that represent a single value — in particular, strings, Booleans, and numbers. But sometimes we need a way to store a *collection* of data. For this, we need data structures. In this and the next lesson, we will learn about a very useful data structure: `Array`s.

Be sure to follow along with the examples in **replit** (https://replit.com/languages/javascript). Remember that you can see the value of variables and other expressions by either using a `console.log` in the code window or entering them directly in the console window.

Identify Data Structures and Arrays

A *data structure* is a means for associating and organizing information. Outside of the programming world, we use data structures all the time. For example, we might have a shopping list of the items we need to buy on our next grocery run or an address book for organizing contact information.

If we have a lot of related data, it's best to represent it in a related system. Imagine that we're working on a lottery application that has to represent the winning lottery numbers. We could do that as follows:

```
const firstNumber = 32;
const secondNumber = 9;
const thirdNumber = 14;
const fourthNumber = 33;
const fifthNumber = 48;
const powerBall = 5;
```

We've represented all six pieces of data, but there's no way to refer to them as a group. Every single time we want to reference that combination of winning numbers, we need to remember and type out six different variable names:

```
const firstNumber = 32;
const secondNumber = 9;
const thirdNumber = 14;
const fourthNumber = 33;
const fifthNumber = 48;
const powerBall = 5;

function logWinningNumbers(first, second, third, fourth, fifth, power) {
  console.log("Winning numbers:", first, second, third, fourth, fifth, power);
}

logWinningNumbers(
  firstNumber,
  secondNumber,
  thirdNumber,
  fourthNumber,
  fifthNumber,
  powerBall
);
```

```
// LOG: Winning numbers: 32 9 14 33 48 5
// => undefined
```

That's so much typing! There are much, much better ways to organize data in JavaScript. Let's learn about one of the most common: the *Array* .

Create Arrays

An `Array` is a list, with the items listed in a particular order, surrounded by square brackets (`[]`) and separated by commas:

```
["This", "is", "an", "array", "of", "strings."];
// => ["This", "is", "an", "array", "of", "strings."]
```

The *members* or *elements* in an `Array` can be data of any type in JavaScript:

```
["Hello, world!", 42, null, NaN];
// => ["Hello, world!", 42, null, NaN]
```

> **NOTE**: In some other languages `Array` s *cannot include elements of multiple types*. In C, C++, Java, Swift, and others you cannot mix types. JavaScript, Python, Ruby, Lisp, and others permit this.

Arrays are *ordered*, meaning that the elements in them will always appear in the same order. This also means that the `Array` `[1, 2, 3]` is different from the `Array` `[3, 2, 1]` .

Just like any other type of JavaScript data, we can assign an `Array` to a variable:

```
const primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37];

const flowers = ["Rose", "Tulip", "Orchid", "Lily"];
```

We can find out how many elements an `Array` contains by checking the `Array` 's built-in `length` property:

```
const myArray = ["This", "array", "has", 5, "elements"];

myArray.length;
// => 5
```

We defined the above `Array`s using the *array literal* syntax — that is, we literally typed out the `Array` that we wanted to create, square brackets and all. There are other ways to create new `Array`s, but they are only necessary for very rare circumstances. For now, use `Array` literals.

To get a sense of just how effective `Array`s are at keeping data organized, let's rewrite our lottery code to use an `Array`:

```
const winningNumbers = [32, 9, 14, 33, 48, 5];

function logWinningNumbers(numbers) {
  console.log("Winning numbers:", numbers);
}

logWinningNumbers(winningNumbers);
// LOG: Winning numbers: [32, 9, 14, 33, 48, 5]
// => undefined
```

The `Array` provides organization, and we only have to remember *one* identifier (`winningNumbers`) instead of six (`firstNumber`, `secondNumber`, and so on). We can also call `Array`s *expressive* because putting all the winning numbers in a shared data structure communicates to other programmers "Hey, these things go together."

The one benefit of storing all six lottery numbers separately is that we had a really easy way to access each individual number. For example, we could just reference `powerBall` to grab the sixth number. Luckily, `Array`s offer an equally simple syntax for accessing individual members.

Using Bracket Notation

Every element in an `Array` is assigned a unique index value that corresponds to its place within the collection, **starting at 0**. The first element in the `Array` is at index `0`, the fifth element at index `4`, and the 428th element at index `427`. We can use **bracket notation** (`[]`) to access

the element at a given index.

Accessing an Element

To access an element, we use bracket notation like this: `nameOfArray[index]`.

```
const winningNumbers = [32, 9, 14, 33, 48, 5];
// => undefined

winningNumbers[0];
// => 32

winningNumbers[3];
// => 33
```

Let's take a minute to think about how we could access the **last** element in any `Array`.

If `myArray` contains 10 elements, the final element will be at `myArray[9]`. If `myArray` contains 15000 elements, the final element will be at `myArray[14999]`. So the index of the final element is always one less than the number of elements in the `Array`. If only we had an easy way to figure out how many elements are in the `Array` ...

```
const alphabet = [
  "a",
  "b",
  "c",
  "d",
  "e",
  "f",
  "g",
  "h",
  "i",
  "j",
  "k",
```

```
    "l",
    "m",
    "n",
    "o",
    "p",
    "q",
    "r",
    "s",
    "t",
    "u",
    "v",
    "w",
    "x",
    "y",
    "z",
];
// => undefined

alphabet.length;
// => 26

alphabet[alphabet.length - 1];
// => "z"
```

We placed an expression (`alphabet.length - 1`) inside the square brackets, and the JavaScript engine *computed* the value of that expression to determine which element we were trying to access. In this case, `alphabet.length - 1` evaluated to `25` , so `alphabet[alphabet.length - 1]` became `alphabet[25]` .

Updating the Value of an Element

We can also use bracket notation (`[]`) — along with the assignment operator (`=`) — to update the value of an element in the array.

Say we've defined an array, `planets` :

```
const planets = [
  "Mercury",
  "Venus",
  "Earth",
  "Mars",
  "Juptier",
  "Saturn",
  "Uranus",
  "Neptune",
];
//=> undefined
```

Looks like we've misspelled Jupiter. Let's fix it. To do that, we access the element we want (`planets[4]`), then reassign that element's value using `=` :

```
planets[4] = "Jupiter";
//=> "Jupiter"

planets;
//=> ["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"]
```

You might wonder why that worked, given that we declared `planets` using `const` . After all, declaring with `const` is supposed to prevent us from reassigning the value of the variable! This is an important thing to know about how `const` works with `Array` s. We are not able to reassign the array *itself* but we **can** reassign any of its elements. We can also add or delete elements. JavaScript `Object` s work similarly when declared using `const` (which isn't surprising, given that `Array` s are a special type of `Object`). The only thing `const` prevents us from doing is assigning a new value to the variable `planets` using the `=` operator:

```
planets = ["new", "array"];
//=> Uncaught TypeError: Assignment to constant variable.
```

Nested Arrays

We mentioned above that arrays can contain elements of **any** data type; this includes **other** `Array` **s**:

```
const egregiouslyNestedArray = [
  "How",
  ["deep", ["can", ["we", ["go", ["?"], "Pretty"], "dang"], "deep,"], "it"],
  "seems.",
];
```

If you examine the array above, you'll see it contains three elements, the second of which is itself an array.

```
egregiouslyNestedArray[0];
//=> 'How'

egregiouslyNestedArray[1];
//=> [ 'deep', [ 'can', [ 'we', [Array], 'dang' ], 'deep,' ], 'it' ]

egregiouslyNestedArray[2];
//=> 'seems.'
```

So we know we can access the inner array using `egregiouslyNestedArray[1]` , but how do we access the array nested inside *that* array? We simply add another set of brackets:

```
egregiouslyNestedArray[1][0];
//=> 'deep'

egregiouslyNestedArray[1][1];
//=> [ 'can', [ 'we', [ 'go', [Array], 'Pretty' ], 'dang' ], 'deep,' ]

egregiouslyNestedArray[1][2];
//=> 'it'
```

We can continue drilling down in this way, adding another set of brackets for each nested array, until we get to the innermost array:

```
egregiouslyNestedArray[1][1][1][1][1];
//=> ['?']
```

That innermost array contains only one element, so how would we access that?

```
egregiouslyNestedArray[1][1][1][1][1][0];
//=> '?'
```

While it's great that `Array`s allow us to store other `Array`s inside them, this is a terrible way to represent a deeply nested data structure. In general, it is best to keep your `Array`s to *no more than two levels deep*. Two levels is perfect for representing two-dimensional things, like a tic-tac-toe board:

```
const board = [
  ["X", "O", " "],
  [" ", "X", "O"],
  ["X", " ", "O"],
];

board;
// => [["X", "O", " "], [" ", "X", "O"], ["X", " ", "O"]]
```

The cool thing about representing a game board like that is in how we can access the different squares by specifying coordinates. The first `[]` operator grabs the **row** that we want, top (`board[0]`), middle (`board[1]`), or bottom (`board[2]`). For example:

```
board[1];
// => [" ", "X", "O"]
```

The second `[]` operator specifies the column, or the square within that row: left (`board[1][0]`), middle (`board[1][1]`), or right (`board[1][2]`). For example:

```
board[0][0];
// => "X"
```

```
board[0][2];
// => " "

board[2][2];
// => "O"
```

Effectively, we're using X and Y coordinates to refer to data within a two- dimensional structure.

Array Methods

JavaScript includes a number of built-in `Array` *methods*, functions that *belong to* (i.e., can only be called on) `Array` s. These methods allow us to manipulate arrays in various ways, for example, to add an element to the beginning or end of an `Array` .

We have seen methods already in this course. In an earlier lesson, for example, we learned how to use a `String` method, `toUpperCase()` , to get the uppercase version of a `String` :

```
const string = "Hello";

string.toUpperCase();
//=> "HELLO"
```

As shown above, we call `String` methods *on* a string using dot notation. `Array` methods work in the same way. Some of them will take one or more arguments — for example, the element we want to add to the array. The arguments are passed in the parentheses after the method name.

We will learn about these `Array` methods in the next lesson. Before we get to that, however, we need to introduce one more concept.

Mutability

Some methods update or *mutate* the object they are called on; these methods are referred to as *destructive*. Other methods, known as *nondestructive* methods, leave the object intact. For example, the `String` method `toUpperCase()` is *nondestructive*:

```
const string = "Hello";

string.toUpperCase();
//=> "HELLO"

string;
//=> "Hello"
```

The `toUpperCase()` method returns the uppercased version of `string`, it does not *change* its value to uppercase.

Sometimes we want to mutate the original object but, in general, it's good practice to avoid mutating a program's state whenever possible. Otherwise, we won't always know what we're dealing with. You will learn more about immutability and why it's important later in the course. But, in the meantime, it's important to know which JavaScript methods are destructive and which are nondestructive so you can choose the appropriate method for what you need to accomplish.

Conclusion

In this lesson, we learned about JavaScript `Array`s, including how to create them, access their elements, and change the value of elements. We also learned that arrays can contain elements of any data type, including other `Array`s; we refer to these as *nested* arrays. Finally, we talked a little bit about the concept of mutability. Armed with this knowledge, we are now ready to tackle JavaScript `Array` methods.

Resources

- **MDN: Array reference (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)**