Array Methods

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

Learning Goals

- Add elements to an `Array`
- Remove elements from an `Array`
- Replace elements in an `Array`

Introduction

In the last lesson, we learned about JavaScript `Array`s, including how to create them and access their elements. In this lab, we will dive into JavaScript's `Array` methods, which enable us to add, remove, and change elements.

We discussed the fact that it's important to pay attention to whether the method is *destructive* (i.e., it *mutates* the array) or *nondestructive*. Another factor to pay attention to is what the *return value* of each of these methods is. Be sure to follow along and experiment with each method in **replit** (https://replit.com/languages/javascript) until you understand how it works, what it does to the original array, and what it returns.

Add Elements to an Array

We'll start with the JavaScript methods we can use to add elements to an array: `.push()` and `.unshift()`, which are *destructive* methods, and the spread operator, which is *nondestructive*.

`.push()` and `.unshift()`

These two methods work in the same way:

- They take one or more arguments (the element or elements you want to add)
- They *return* the length of the modified array

- They are *destructive* methods

The difference is that the `.push()` method adds elements to the end of an `Array` and `.unshift()` adds them to the beginning of the array:

```
const superheroes = ["Catwoman", "Storm", "Jessica Jones"];

superheroes.push("Wonder Woman");
// => 4

superheroes;
// => ["Catwoman", "Storm", "Jessica Jones", "Wonder Woman"]

const cities = ["New York", "San Francisco"];

cities.unshift("Boston", "Chicago");
// => 4

cities;
// => ["Boston", "Chicago", "New York", "San Francisco"]
```

Before moving on, try out the examples above as well as some examples of your own in the REPL.

Spread Operator

The *spread operator*, which looks like an ellipsis: `...`, allows us to "spread out" the elements of an existing `Array` into a new `Array`, creating a copy:

```
const coolCities = ["New York", "San Francisco"];

const copyOfCoolCities = [...coolCities];
```

```
copyOfCoolCities;
//=> ["New York", "San Francisco"]
```

You might wonder why we would do this rather than just `copyOfCoolCities = coolCities`. The answer is that `coolCities` *points to a location in memory* and when you use the assignment operator to create a copy, you create a second variable that points to the *same* location. What this means is that, if you change `copyOfCoolCities`, `coolCities` is changed as well (and vice versa).

> **A note about copying arrays in JavaScript** Copying arrays in JavaScript is complicated! Some methods of copying create *deep* copies and some create *shallow* copies. Using the spread operator to copy an array creates a shallow copy. What this means is that, if you use it to copy a nested array, the inner array (or arrays) *still points to the same location in memory* as in the original array. So if you modify the *inner* array in the copy, it changes the inner array in the original array as well (and vice versa). Don't worry too much about shallow and deep copies at this point: just know that you can safely use the spread operator to clone *non-nested* arrays.

Because the spread operator is an operator rather than a method, it works differently than `push()` and `unshift()`: in the example above, we're constructing an `Array` using *literal* notation (i.e., typing the square brackets) and populating it by using the spread operator on the `Array` we want to copy.

Where the spread operator comes in especially handy is when we want to add one or more new elements either before or after the elements in the original array (or both) without mutating the original array. To add an element to the front of the new array, we simply type in the new element before spreading the elements in the original array:

```
const coolCities = ["New York", "San Francisco"];

const allCities = ["Los Angeles", ...coolCities];

coolCities;
// => ["New York", "San Francisco"]

allCities;
// => ["Los Angeles", "New York", "San Francisco"]
```

And, as you might expect, to add a new item to the end of an `Array`, we type in the new element *after* spreading the elements in the original array:

```
const coolCats = ["Hobbes", "Felix", "Tom"];

const allCats = [...coolCats, "Garfield"];

coolCats;
// => ["Hobbes", "Felix", "Tom"]

allCats;
// => ["Hobbes", "Felix", "Tom", "Garfield"]
```

Note that, in both cases, we created a new `Array` instead of modifying the original one — our `coolCities` and `coolCats` `Array`s were untouched. Because we didn't modify the original array, in order to save the results of our work we had to assign it to a variable.

Be sure to experiment with the spread operator in **replit (https://replit.com/languages/javascript)** until you're comfortable with how it works — it will come in handy later!

Remove Elements from an Array

As complements for `.push()` and `.unshift()`, respectively, we have `.pop()` and `.shift()`.

`.pop()` and `.shift()`

As with `.push()` and `.unshift()`, these two methods work in the same way:

- they don't take any arguments
- they remove a single element
- they *return* the element that is removed
- they are *destructive* methods

The `.pop()` method removes the last element in an `Array`:

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.pop();
// => "Sun"

days;
// => ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
```

The `.shift()` method removes the first element in an `Array`:

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.shift();
// => "Mon"

days;
// => [Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

Notice above that both methods returned the removed element and mutated the original array.

`.slice()`

To remove elements from an `Array` nondestructively (without mutating the original `Array`), we can use the `.slice()` method. Just as the name implies, `.slice()` returns a portion, or **slice**, of an `Array`. The method takes 0, 1, or 2 arguments and returns a new array containing the sliced elements.

With No Arguments

If we don't provide any arguments, `.slice()` will return a copy of the original `Array` with all elements intact:

```
const primes = [2, 3, 5, 7];
```

```
const copyOfPrimes = primes.slice();

primes;
// => [2, 3, 5, 7]

copyOfPrimes;
// => [2, 3, 5, 7]
```

Note that creating a copy using `.slice()` works the same way as if you use the spread operator: they both create a *shallow* copy, and with both the copy points to a different object in memory than the original. If you add an element to one of the arrays, it does **not** get added to the others:

```
const primes = [2, 3, 5, 7];

const copyOfPrimesUsingSlice = primes.slice();

const copyOfPrimesUsingSpreadOperator = [...primes];

primes.push(11);
// => 5

primes;
// => [2, 3, 5, 7, 11]

copyOfPrimesUsingSlice;
// => [2, 3, 5, 7]

copyOfPrimesUsingSpreadOperator;
// => [2, 3, 5, 7]
```

With Arguments

We can also provide one or two arguments to `.slice()` : the first is the index where the slice should begin and the second is the index **before which** it should end:

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.slice(2, 5);
// => ["Wed", "Thu", "Fri"]
```

If no second argument is provided, the slice will run from the index specified by the first argument to the end of the `Array` :

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.slice(5);
// => ["Sat", "Sun"]
```

To return a new `Array` with the first element removed, we call `.slice(1)` :

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.slice(1);
// => ["Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

And we can return an array with the last element removed in a way that will look familiar from the previous lesson:

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.slice(0, days.length - 1);
// => ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
```

However, `.slice()` provides an easier syntax for referencing the last element (or elements) in an `Array` :

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.slice(-1);
// => ["Sun"]

days.slice(-3);
// => ["Fri", "Sat", "Sun"]

days.slice(0, -1);
// => ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
```

When we provide a negative index, the JavaScript engine knows to start counting from the last element in the `Array` instead of the first.

.splice()

Unlike `.slice()`, which is nondestructive, `.splice()` performs destructive actions. Depending on how many arguments we give it, `.splice()` allows us to remove elements, add elements, or replace elements (or any combination of the three).

With a Single Argument

```
array.splice(start);
```

The first argument expected by `.splice()` is the index at which to begin the splice. If we only provide the one argument, `.splice()` will destructively remove a chunk of the original `Array` beginning at the provided index and continuing to the end of the `Array`:

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

days.splice(2);
// => ["Wed", "Thu", "Fri", "Sat", "Sun"]

days;
// => ["Mon", "Tue"]
```

Notice that `.splice()` **both** mutated the original array (by removing a chunk, leaving just `["Mon", "Tue"]`) **and** returned a new array containing the removed chunk.

We can use a negative 'start' index with `splice()`, the same as with `slice()`:

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
// => ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

days.splice(-2);
// => ["Sat", "Sun"]

days;
// => ["Mon", "Tue", "Wed", "Thu", "Fri"]
```

With Two Arguments

```
array.splice(start, deleteCount);
```

When we provide two arguments to `.splice()`, the first is still the index at which to begin splicing, and the second dictates how many elements we want to remove from the `Array`. For example, to remove `3` elements, starting with the element at index `2`:

```
const days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
// => ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

days.splice(2, 3);
// => ["Wed", "Thu", "Fri"]

days;
// => ["Mon", "Tue", "Sat", "Sun"]
```

Here again, we see that `splice()` removed elements from the original array, mutating that array, and returned the removed elements in a new array.

Replace Elements in an Array

`.splice()` with 3+ arguments

```
array.splice(start, deleteCount, item1, item2, ...)
```

After the first two, every additional argument passed to `.splice()` will be inserted into the `Array` at the position indicated by the first argument. We can replace a single element in an `Array` as follows, discarding a card and drawing a new one:

```
const cards = [
  "Ace of Spades",
  "Jack of Clubs",
  "Nine of Clubs",
  "Nine of Diamonds",
  "Three of Hearts",
];

cards.splice(2, 1, "Ace of Clubs");
// => ["Nine of Clubs"]

cards;
// => ["Ace of Spades", "Jack of Clubs", "Ace of Clubs", "Nine of Diamonds", "Three of Hearts"]
```

We have deleted "Nine of Clubs" and inserted "Ace of Clubs" in place, effectively *replacing* the original card.

Or we can remove two elements and insert three new ones as our restaurant expands its vegetarian options:

```
const menu = [
  "Jalapeno Poppers",
  "Cheeseburger",
  "Fish and Chips",
  "French Fries",
```

```
  "Onion Rings",
];

menu.splice(1, 2, "Veggie Burger", "House Salad", "Teriyaki Tofu");
// => ["Cheeseburger", "Fish and Chips"]

menu;
// => ["Jalapeno Poppers", "Veggie Burger", "House Salad", "Teriyaki Tofu", "French Fries", "Onion Rings"]
```

We aren't required to remove anything with `.splice()` — we can use it to insert any number of elements anywhere within an `Array` by passing 0 as the second argument. Here we're adding new books to our library in alphabetical order:

```
const books = ["Beloved", "Giovanni's Room", "The Color Purple", "The Grass Dancer"];

books.splice(2, 0,  "Kindred", "Love Medicine");
// => []

books;
// => ['Beloved', 'Giovanni's Room', 'Kindred', 'Love Medicine', 'The Color Purple', 'The Grass Dancer']
```

Notice that `.splice()` returns an empty `Array` when we provide a second argument of `0`. This makes sense because the return value is the set of elements that were removed, and we're telling it to remove `0` elements.

Keep playing around with `.splice()` in the REPL to get comfortable with it.

Using Bracket Notation to Replace Elements

Recall from the previous lesson that we can also use bracket notation to replace a single element in an `Array`. If we only need to replace one element, this is a simpler approach:

```
const cards = [
  "Ace of Spades",
  "Jack of Clubs",
```

```
  "Nine of Clubs",
  "Nine of Diamonds",
  "Three of Hearts",
];

cards[2] = "Ace of Clubs";
// => "Ace of Clubs"

cards;
// => ["Ace of Spades", "Jack of Clubs", "Ace of Clubs", "Nine of Diamonds", "Three of Hearts"]
```

Both this approach and `splice()` are destructive — they modify the original `Array`. There's a *nondestructive* way to replace or add items at arbitrary points within an `Array`; to do it we need to combine the `slice()` method and the spread operator.

Slicing and Spreading

Combining `.slice()` and the spread operator allows us to replace elements *nondestructively*, leaving the original `Array` unharmed:

```
const menu = [
  "Jalapeno Poppers",
  "Cheeseburger",
  "Fish and Chips",
  "French Fries",
  "Onion Rings",
];

const newMenu = [
  ...menu.slice(0, 1),
  "Veggie Burger",
  "House Salad",
  "Teriyaki Tofu",
  ...menu.slice(3),
];
```

```
menu;
// => ["Jalapeno Poppers", "Cheeseburger", "Fish and Chips", "French Fries", "Onion Rings"]

newMenu;
// => ["Jalapeno Poppers", "Veggie Burger", "House Salad", "Teriyaki Tofu", "French Fries", "Onion Rings"]
```

Let's unpack this a little bit. We're assigning an array to `newMenu` using literal notation. Inside the brackets, we are spreading the result of calling slice on `menu` with the arguments `0` and `1`, then typing in three new elements, then spreading the result of calling slice on `menu` with the argument `3`. Here, we are taking advantage of the fact that the `slice()` method **returns a new array**. We can spread the elements in **that** array just as we can with any other array.

Play around with this in the REPL until it makes sense; break it down into its component parts and try each piece on its own. It's the trickiest thing that we've encountered in this lesson, so don't sweat it if it takes a little while to sink in!

Conclusion

In this lesson, we've learned a variety of methods we can use to remove, add, and replace the elements in `Array`s. We've learned that some methods are *destructive* and some are *nondestructive*. With this knowledge, you have the tools you need to manipulate `Array`s in very complex ways.

JavaScript Array Methods

Method	What it does	Arguments	Destructive/ Nondestructive	Return value
Spread operator (...)	Add element(s) to beginning or end of array	N/A	Nondestructive	The new array
.push()	Add element(s) to end of array	1 or more	Destructive	The length of the modified array
.unshift()	Add element(s) to beginning of array	1 or more	Destructive	The length of the modified array
.pop()	Remove last element	0	Destructive	The removed element
.shift()	Remove first element	0	Destructive	The removed element
.slice()	Remove 0 or more elements	0, 1, or 2	Nondestructive	A new array containing the removed elements
.splice()	Remove, add, or replace elements	1 or more	Destructive	A new array containing the removed elements

Resources

- MDN
  - Array (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)
  - `.slice()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice)
  - `.splice()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice)