Front-End Web Development

Arrays and Loops

GENERAL ASSEMBLY

Today's Learning Objectives

In this lesson, you will:

- Use arrays and loops in JavaScript to manage collections of data.
- Invoke array methods to manipulate the contents of an array.
- Distinguish between `for` loops and `while` loops.

Arrays

Think of arrays as containers of data. Technically, an **array** is an ordered collection of data types combined into one variable.

Each item in an array is assigned an **index** value based on its position. These index values allow us to access individual elements within the array.

```
["banana", "orange", "apple"]
  0         1         2
```


Why Do Arrays Matter?

Arrays are one of just two types of data “containers” in JavaScript (the other is objects). As a result, much of the data we use comes in the form of arrays.

Arrays come from three main places:

1. The DOM (with `querySelectorAll` and `getElementsByClassName`)
2. API responses (information received from other web applications)
3. Databases

Syntax

```
const fruits = ["banana", "orange", "apple"]
```

You make an array using a set of square brackets.

Inside the brackets, each value must be separated by a comma.

Accessing Array Values

```
const fruits = ["banana", "orange", "apple"];
fruits[0]; // will output "banana"
fruits[1]; // will output "orange"
fruits[2]; // will output "apple"
```

Access array items using square brackets around their index values. It's pretty simple — just remember that the first index value is always zero!

Front-End Web Development

Array Properties and Methods

Guided Walk-Through:
Arrays Reference Code

You can follow along with array methods and properties in this CodePen:

Reference code:

https://codepen.io/GAmarketing/pen/RwNVvoW

.length

```
const fruits = ["banana", "orange", "apple"];

fruits.length;
=> 3
```

Use the **.length** property to figure out how many items are in your array. This is very useful when we need to look through the whole array with a loop.

.indexOf()

```
const fruits = ["banana", "orange", "apple"];

fruits.indexOf("orange");
// will return the number 1
```

Use **.indexOf()** to see the index value of any item in the array.

Adding Items With `.push`

```
const fruits = ["orange"];

fruits.push("kiwi");
// fruits is now: ["orange", "kiwi"];
```

The **`.push`** method adds the item inside the parentheses to the **end** of the array.

Removing Items With `.pop`

```
const fruits = ["banana", "orange", "apple"];

fruits.pop();
// fruits is now: ["banana", "orange"];
```

The **`.pop`** method removes the **last** item in the array. Can you hear the sound effect when you pop an item off?

Adding Items With `.unshift`

```
const fruits = ["orange", "kiwi"];

fruits.unshift("cherry");
// fruits is now: ["cherry", "orange", "kiwi"];
```

The **`.unshift`** method adds the item inside the parentheses to the **beginning** of the array.

Removing Items With `.shift`

```
const fruits = ["banana", "orange"];

fruits.shift();
// fruits is now: ["orange"];
```

The **`.shift`** method removes the **first** item in the array. This one just quietly shifts off into the night.

Removing Items With `.splice`

```
const fruits = ["cherry", "orange", "kiwi"];

fruits.splice(1,1);
// fruits is now: ["cherry", "kiwi"];
```

The **`.splice`** method removes specific items from the array.

The first number indicates the index where removal begins, and the second number indicates the total number of items to remove.

Adding Items With `.splice`

```
fruits = ["cherry", "kiwi"];

fruits.splice(1, 0, "pear");
// 1st value = index value for splice
// 2nd value = number of items to remove
// 3rd value = item to be added to array
// fruits is now: ["cherry", "pear", "kiwi"];
```

If you use a third value in the **`.splice`** method, it will **add** that value into your array at the location indicated by the first value.

.reverse() Order

```
const fruits = ["cherry", "pear", "kiwi"];

const fruits.reverse();
// fruits is now: ["kiwi", "pear", "cherry"];
```

True to its name, the **.reverse** method reverses the order of all items in the array. It doesn't add or delete anything.

Combining Elements With `.join`

```
const fruits = ["kiwi", "pear", "cherry"];

const fruitsList = fruits.join(" and ");
// fruitsList is: "kiwi and pear and cherry";
```

`.join` combines all the items in the array together into a string. If given a string as a parameter, **`.join`** will place the given string in between the elements.

Multi-Dimensional Arrays (Matrix)

```
const produce = [["kiwi", "pear"], ["carrots", "celery"]];

// produce[1] is: ["carrots”, “celery”];
// produce[0][1] is: "pear";
```

You can even put arrays inside of arrays! Access two-dimensional arrays using a second set of square brackets: The first brackets locate your desired array, and the second brackets locate the item within that array.

Guided Walk-Through:
Interactive Fruit Basket

We can see an interactive version of these examples in this CodePen, along with a very different way to perform DOM manipulations: the jQuery library!

Reference code:

https://codepen.io/GAmarketing/pen/jOEmdJv

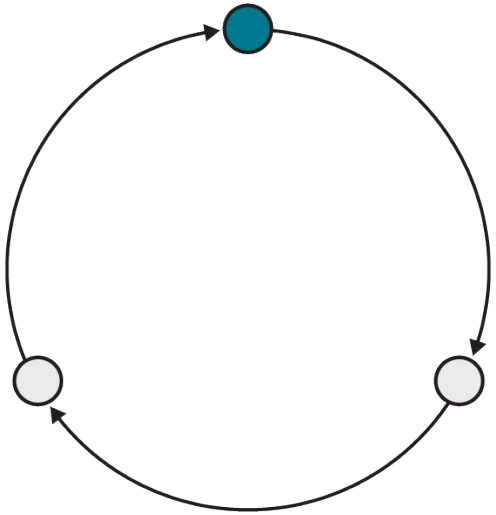
Front-End Web Development

Loops

Loop: A control flow statement allowing for the repeated execution of a code block until a specific condition is reached.

Why Loops?

- **Loops** take advantage of what computers do best: evaluate instructions across organized sets of data very quickly.
- Computers excel when working in isolated patterns, which is exactly how a loop works.
- Avoid needlessly copying or re-typing code by repeating it in a loop.



An Iterator, Terminator, and Incrementer Walk Into a Loop...

A **`for loop`** is similar to an **`if`** statement but with more conditions. When creating a **`for`** loop, we need to make three declarations:

1. Define a variable to act as our **iterator**, typically named **`i`**.
2. Establish a condition for the loop to stop, called the terminating condition.
3. Increment the iterator variable (or decrement, if the loop goes backward).

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
// outputs 0,1,2,3,4,5,6,7,8,9
```


Front-End Web Development

Loops + Arrays: A Classic Combination

`for` Loop + Arrays

Notice the highlighted terminating condition. The array's length limits the number of loops.

```
const myArray = ["John", "Benjamin", "Victor", "Serrao"];

for (let i = 0; i < myArray.length; i++) {
  console.log(myArray[i]);
}
// outputs each name in the list, one at a time
```

`while` Loop

The **`while`** loop, a simpler cousin of the **`for`** loop, will run as long as a condition remains true.

```
const myArray = ["John", "Benjamin", "Victor", "Serrao"];
let i = 0;

while (i <= myArray.length) {
  i++;
}
```


`for` vs. `while` Loops

These loops seem so similar — which one should you choose? Most of the time, the answer is a `for` loop.

`for` loops are perfect for when you have a specific number of times you need the loop to run. In most cases, that number is the length of an array.

A `while` loop is useful when there's no way of knowing how many times the loop will need to repeat — like a game loop that stops when a character dies.

Guided Walk-Through:
Loops Reference Code

The following CodePen shows how loops can be used to display lists of information:

Reference code:

https://codepen.io/GAmarketing/pen/NWPjJGL

Front-End Web Development

return-ing to Functions

return

Programs with multiple, reusable functions that process and `return` data are the next step in your journey.

```
function addThings(val1, val2) {
  return val1 + val2;
}

let result = addThings(1, 2);
console.log(result);
```


return + Conditions

Functions will often have conditions that change the nature of their output.

```
function addThings(val1, val2) {
  if (val1 >= 10) {
    return val1 + val2;
  }
  return 0;
}

console.log(addThings(11, 2));
console.log(addThings(7, 5));
```


Practical Application

These examples may seem kind of silly (they are!), but that's because our data is static.

Loops and functions become more important when we handle **live data** from APIs and databases, because they help us deal with information we haven't defined ourselves!

Key Takeaways

Use Loops to Handle Arrays

- Arrays can contain any amount of any data type.
- Loops create dynamic, DRY code for iterating.
- Use `for` loops for a specific number of iterations and `while` loops for unknown quantities.

For Next Time

Forms

- We'll use forms to collect data from users, validate the information they submit, and display error messages where they have submitted invalid values.


GA