2.03 Array Iteration.

Objectives

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

- Understand what iteration is and why it is useful
- Compare and contrast `for`, `while` and `do...while` loops
- Use `break` and `continue` to write more efficient loops

Very often, you'll want to access each element of an array in order and do something with each element. For example, maybe you have an array of tweets, and you want to show each one on the page. Or maybe you have a list of numbers that you want to apply some mathematical operation to.

For instance, suppose you have an array of numbers that you want to round to the nearest whole number:

```
let decimals = [1.1, 1.6, 2.8, 0.4, 3.5, 1.6];
```

One way to do this is to round each element individually using the built-in `Math.round` function:

```
decimals[0] = Math.round(decimals[0]);
decimals[1] = Math.round(decimals[1]);
decimals[2] = Math.round(decimals[2]);
decimals[3] = Math.round(decimals[3]);
decimals[4] = Math.round(decimals[4]);
decimals[5] = Math.round(decimals[5]);
```

Now if you look at `decimals`, you should see that it is equal to `[1, 2, 3, 0, 4, 2]`. Great! We've rounded all of our numbers.

But this approach isn't great. What if we have 100 numbers we want to round? Or 1,000? And what if we want to do something more complicated than simply round each one? The approach we've used here doesn't scale very well.

Thankfully, there's a better way to make your way through an array and do something with each element, through a process called *iteration*, or looping. Let's talk about iteration in general, and then see how we can apply it to arrays.

Iteration: For loops

One of the most common ways to loop is with a `for` loop. A `for` loop consists of three parts followed by a block of code inside of curly braces `{}`:

```
for (initializer; condition; counter) {}
```

`initializer` - this is where we can declare variables to be used in the loop. We usually declare a variable called `i` which will serve as a counter variable for the number of times that we should loop.

`condition` - this MUST be an expression that returns `true` or `false`. You can read this condition as "Keep looping as long as this condition is true."

`counter` - this is how we change the variables initialized (typically, either by increasing or decreasing them). We commonly increment variables by 1 using the `++` operator and decrement by 1 using `--`.

As long as the condition is true, the code inside the curly braces will run. After running, the counter expression will run, and then the condition will be checked again.

```
// start with a variable called i and set it to 0
// keep looping as long as i is less than 5
// at the end of each for loop, increase the value of i
for (let i=0; i<5; i++) {
    console.log(i);
}

// prints out:

// 0
// 1
// 2
// 3
// 4
```

What gets logged if you change `i<5` to `i<10`? If you change `i++` to `i+=3`? Experimenting with the initializer, condition, and counter is a great way to develop your intuition for `for` loops!

You can use a loop to iterate through an array in a similar fashion. In this case, typically `i` refers to the current index in the array, the `condition` tells the loop to continue until `i` equals the length of the array, and the `counter` increments `i`. Let's see how we could refactor our earlier rounding example to use a for loop:

```
let decimals = [1.1, 1.6, 2.8, 0.4, 3.5, 1.6];

for (let i = 0; i < decimals.length; i++) {
    decimals[i] = Math.round(decimals[i]);
}
```

Iteration: While loops

Along with `for` loops, we can also use a `while` loop. Unlike `for` loops, `while` loops only take a `condition`. This means that you need to handle initialization before the loop, and incrementing/decrementing yourself inside of the loop. If you forget to increment/decrement inside the loop, the loop will never terminate! Instead, you'll be stuck in what's called an *infinite loop*!

Here's an example of a working `while` loop:

```
let i = 0;
while (i < 5) {
    console.log(i);
    i++;
}
```

Here's how we could rewrite our rounding example to use a `while` loop:

```
let decimals = [1.1, 1.6, 2.8, 0.4, 3.5, 1.6];
let i = 0;

while (i < decimals.length) {
    decimals[i] = Math.round(decimals[i]);
    i++;
}
```

Iteration: Do While Loops

Similar to while loops, we can also write `do...while` loops, which specify our `condition` at the end. Here is an example:

```
let i = 0;
do {
    console.log(i);
    i++;
} while(i < 5)
```

The main difference between a `while` loop and a `do...while` loop is that the code inside of a `do...while` loop is guaranteed to execute at least once. For example:

```
let i = 0;
while(i < 0) {
    console.log(i);
    i++;
}
// nothing is logged, since 0 < 0 is false

let j = 0;
do {
    console.log(j);
    j++;
} while(j < 0)
// 0 gets logged, since the code inside the block runs once
// before the while condition is checked
```

Here's how we could rewrite our rounding example to use a `do...while` loop:

```
let decimals = [1.1, 1.6, 2.8, 0.4, 3.5, 1.6];
let i = 0;

do {
    decimals[i] = Math.round(decimals[i]);
    i++;
} while(i < decimals.length)
```

Exiting out of loops

Sometimes we want to exit a loop before it has finished. To do that, we use the word `break`

```
for(let i = 0; i<5; i++){
    if(Math.random() > 0.5){
        console.log("Breaking out of the loop when i is " + i);
        break;
    }
    else {
        console.log(i);
    }
}
```

We can also skip the current iteration and continue the loop at the next step in the itration by using the word `continue`

```
for(let i = 0; i<5; i++){
    if(Math.random() > 0.5){
        console.log("Skipping the console.log when i is " + i);
        continue;
    }
    console.log(i);
}
```

For...of, a more modern loop

In 2015, JavaScript introduced a simpler kind of loop that allows you to iterate with less code. It's called `for...of` loop, and it looks like this:

```
let names = ["Elie", "Matt", "Tim"];
for (let name of names) {
    console.log(name);
}
```

This will output:

```
// Elie
// Matt
// Tim
```

With a `for...of` loop, we iterate over an array and assign a variable to each element in the array. You can all this variable whatever you like (in our example we called it `name`), just be sure to declare it using `let`.

Let's see another example:

```
let numbers = [2,4,6,8];
for (let num of numbers) {
    console.log(num);
}
```

Here we will log 2, 4, 6, and 8.

If you need to access the index of the array, you *can* do this using a `for...of` loop, but it's a bit complex. For now, if you need each index, use a regular `for` or `while` loop.

Strings Revisited

Now that we've learned about arrays, let's briefly return to strings and compare and contrast these two data types. They do have some similarities, but it's important to understand their differences as well.

Looping over strings

Just like we can iterate over arrays (and objects), we can also iterate over strings! Since strings have a `length` property, we always know at what point to stop looping, just like with arrays. Let's see an example of looping over a string:

```
let name = "Elie";

for(let i=0; i < name.length; i++){
    console.log(name[i]);
}

// E
// l
// i
// e
```

You can also use a `for...of` to loop over characters in a string!

```
let name = "Kayla";

for (let character of name) {
    console.log(character);
}

// K
// a
```

```
// y
// l
// a
```

Using split to turn a string into an array

Many times you will need to manipulate a string and turn it into an array. To split a string into an array you can use the `split` method and pass in a delimiter value.

```
let string = "hello world";
string.split(""); // ["h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d"]
string.split(" ");  // ["hello", "world"]
```

If you pass a delimiter into the `split` method, the delimiting values will be removed from the array:

```
let dashedString = "lots-of-dashes-here";
let removedDashes = dashedString.split("-");
removedDashes; // ["lots", "of", "dashes", "here"]
```

We can then join the array using the `join` method to bring it back to a string. You can think of the `split` as doing the opposite of what `join` does.

```
let dashedString = "lots-of-dashes-here";
let removedDashes = dashedString.split("-").join(" ");
removedDashes; // "lots of dashes here"
```

Mutability

We've seen how you can update array values by simply accessing an array element and assigning it a new value:

```
let arr = ["hi", "bye"];
arr[0] = "hello";
arr; // ["hello", "bye"]
```

You can also access characters in strings using bracket notation:

```
let name = "Matt";
name[0]; // "M"
```

However, unlike with arrays, you can't reassign the value of a character in a string. If you try, JavaScript will simply ignore you:

```
let name = "Matt";
name[0] = "m";
name; // "Matt", not "matt"!
```

This distinction between arrays and strings highlights a concept called *mutability*. We say that arrays in JavaScript are mutable, since you can change any element inside of them via a simple reassignment.

However, strings are *immutable*, as you cannot change the characters within them in the same way that you do with arrays. In fact, any operation which changes characters in a string actually produces a new string, rather than mutating the original string.

For more on mutability in JavaScript, you may want to check out this article. Note: the article makes use of functions in JavaScript, so it may be best to read it after finishing the functions unit in this course.

Exercises

The next chapter has plenty of array exercises for you to tackle!