Object Iteration

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

Learning Goals

- Explain the difference between looping and iteration.
- Iterate over arrays with the `for...of` statement.
- Enumerate an object's properties with the `for...in` statement.

Introduction

When we create a `for` loop to loop over an array, we base the loop's condition off of the `.length` of the array. This works, but it's a lot of syntactic cruft to remember:

```
for (let i = 0; i < array.length; i++) {
  // Loop body
}
```

The problem is that we're using a *looping* construct to perform *iteration*.

Looping vs. Iteration

There's a pretty fine line separating the concepts of *looping* and *iteration*, and only the truly pedantic will call you out if you use one in place of the other.

Looping is the process of executing a set of statements **repeatedly until a condition is met**. It's great for when we want to do something a specific number of times (`for` loop) or unlimited times until the condition is met (`while` or `do while` loop).

Iteration is the process of executing a set of statements **once for each element in a collection**. We can accomplish this with a `for` loop:

```
let myArray = ['a', 'b', 'c', 'd', 'e', 'f', 'g'];

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

or with a `while` loop:

```
let myArray = ['a', 'b', 'c', 'd', 'e', 'f', 'g'];

let j = 0;

while (j < myArray.length) {
  console.log(myArray[j++]);
}
```

but neither is very pretty. The `for...of` statement gives us a better way.

`for...of`

Using `for...of`, the code above becomes:

```
const myArray = ['a', 'b', 'c', 'd', 'e', 'f', 'g'];

for (const element of myArray) {
  console.log(element);
}
```

Using a construct that is specifically meant for iteration results in much cleaner code: there's no initialization of a counter, no condition, no incrementing the counter, and no bracket notation to access elements in the array (`myArray[i]`).

`const` vs. `let`

As you might've noticed, `for...of` allows us to use `const` instead of `let`. In `for` and `while` statements, `let` is required because we are incrementing a counter variable. The incrementing process involves taking the counter's current value, adding `1` to it, and then assigning that new value to the variable. That reassignment precludes us from using our beloved `const`, which cannot be reassigned.

Delightfully, the `for...of` statement involves no such reassignment. On each trip into the loop body (which is a *block* — note the curly braces), we assign the next element in the collection to a **new** `element` variable. Upon reaching the end of the block, the block-scoped variable vanishes, and we return to the top. Then we repeat the process, assigning the next element in the collection to a **new** `element` variable.

Iterating over... strings?

A string is effectively an ordered collection (like an array) of characters, which `for...of` is more than happy to iterate over:

```
for (const char of 'Hello, world!') {
  console.log(char);
}

// LOG: H
// LOG: e
// LOG: l
// LOG: l
// LOG: o
// LOG: ,
// LOG:
// LOG: w
// LOG: o
// LOG: r
// LOG: l
// LOG: d
// LOG: !
```

Usage

Use a `for...of` statement anytime you want to iterate over an array.

Iterating over objects

The `for...in` statement is similar to `for...of`; it's generally used for iterating over the properties in an object. The statement follows this syntax:

```
for (const [KEY] in [OBJECT]) {
  // Code in the statement body
}
```

The `for...in` statement iterates over the properties in an object, but it doesn't pass the entire property into the block. Instead, it only passes in the *keys*:

```
const address = {
  street1: '11 Broadway',
  street2: '2nd Floor',
  city: 'New York',
  state: 'NY',
  zipCode: '10004',
};

for (const key in address) {
  console.log(key);
}

// LOG: street1
// LOG: street2
// LOG: city
// LOG: state
// LOG: zipCode
```

Accessing the object's values is as simple as combining the passed-in key with the *bracket operator*:

```
const address = {
  street1: '11 Broadway',
  street2: '2nd Floor',
  city: 'New York',
  state: 'NY',
  zipCode: "10004"
};

for (const key in address) {
  console.log(address[key]);
}

// LOG: 11 Broadway
// LOG: 2nd Floor
// LOG: New York
// LOG: NY
// LOG: 10004
```

But... but I want to use the dot operator!

Can you think of why the bracket operator is required? Let's see what happens when we use the *dot operator*:

```
const address = {
  street1: '11 Broadway',
  street2: '2nd Floor',
  city: 'New York',
  state: 'NY',
  zipCode: '10004'
};
```

```
for (const key in address) {
  console.log(address.key);
}

// LOG: undefined
// LOG: undefined
// LOG: undefined
// LOG: undefined
// LOG: undefined
```

The `for...in` statement iterates over the five properties in `address`, successively passing in the object's keys. However, inside the statement body we're trying to access `address.key`. If you recall from the lesson on objects, variables don't work with the dot operator because it treats the variable name as a literal key — that is, `address.key` is trying to access the property on `address` with a key of `key`. Since there is no `key` property in `address`, it returns `undefined`. To prove this, let's add a `key` property to `address`:

```
address.key = "Let's have a 'key' key!";

for (const key in address) {
  console.log(address.key);
}

// LOG: Let's have a 'key' key!
// LOG: Let's have a 'key' key!
// LOG: Let's have a 'key' key!
// LOG: Let's have a 'key' key!
// LOG: Let's have a 'key' key!
// LOG: Let's have a 'key' key!
```

Usage

Use a `for...in` statement whenever you want to enumerate the properties of an object.

`for...in` and order

Because **arrays are objects**, `for...in` *will work* with arrays. In fact, because `for...of` was added to JavaScript later than `for...in`, you might see older code that uses `for...in` to iterate over arrays. However, as a general rule, **don't use `for...in` with arrays**. When iterating over an array, an **ordered** collection, we would expect the elements in the array to be dealt with **in order**. However, because of how `for...in` works under the hood, there's no guarantee of order. From the **MDN documentation (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in)**:

> A `for...in` loop iterates over the properties of an object in an **arbitrary order** ... one cannot depend on the seeming orderliness of iteration, at least in a cross-browser setting.

What this means is that, with `for...in`, different browsers might iterate over the same object's properties in different orders. That's not cool! Cross-browser consistency is very important. A lot of progress has been made towards standardizing the behavior of `for...in` across all major browsers, but there's still no reason to use `for...in` with arrays when we have the wonderfully consistent `for...of` tailor-made for the job.

Resources

- **MDN — `for...of` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of)**
- **MDN — `for...in` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in)**