Before you move on, let's spend some time looking at two new protocols in ES6:

- the **iterable** protocol
- the **iterator** protocol

These protocols aren't built-ins, but they will help you understand the new concept of iteration in ES6, as well as show you a use case for symbols.

The Iterable Protocol

The **iterable protocol** is used for defining and customizing the iteration behavior of objects. What that really means is you now have the *flexibility* in ES6 to specify a way for iterating through values in an object. For some objects, they already come built-in with this behavior. For example, strings and arrays are examples of built-in iterables.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
for (const digit of digits) {
  console.log(digit);
}
```

```
0
1
2
3
4
5
6
7
8
9
```

If you recall from earlier lesson 1, any object that is iterable can use the new `for...of` loop. Later in this lesson, you'll also learn about Sets and Maps which are other examples of built-in iterables.

How it Works

In order for an object to be iterable, it must implement the **iterable interface**. If you come from a language like Java or C, then you're probably familiar with interfaces, but for those of you who aren't, that basically means that in order for an object to be iterable it must contain a default iterator method. This method will define how the object should be iterated.

The **iterator method**, which is available via the constant `[Symbol.iterator]` , is a zero arguments function that returns an iterator object. An iterator object is an object that conforms to the iterator protocol.

The Iterator Protocol

The **iterator protocol** is used to define a standard way that an object produces a sequence of values. What that really means is you now have a process for defining how an object will iterate. This is done through implementing the `.next()` method.

How it Works

An object becomes an iterator when it implements the `.next()` method. The `.next()` method is a zero arguments function that returns an object with two properties:

1. `value` : the data representing the next value in the sequence of values within the object
2. `done` : a boolean representing if the iterator is *done* going through the sequence of values
    1. If done is *true*, then the iterator has reached the end of its sequence of values.
    2. If done is *false*, then the iterator is able to produce another value in its sequence of values.

Here's the example from earlier, but instead we are using the array's default iterator to step through the each value in the array.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const arrayIterator = digits[Symbol.iterator]();

console.log(arrayIterator.next());
console.log(arrayIterator.next());
console.log(arrayIterator.next());
```

```
Object {value: 0, done: false}
Object {value: 1, done: false}
Object {value: 2, done: false}
```

```
/*
 * Programming Quiz: Make An Iterable Object
 *
 * Turn the `james` object into an iterable object.
 *
 * Each call to iterator.next should log out an object with the following info:
 *   - key: the key from the `james` object
 *   - value: the value of the key from the `james` object
 *   - done: true or false if there are more keys/values
 *
 * For clarification, look at the example console.logs at the bottom of the code.
 *
 * Hints:
 *   - Use `Object.keys()` to store the object's properties in an array.
 *   - Each call to `iterator.next()` should use this array to know which property to return.
```