Arrays and Objects - What is Really Going On?

[(https://github.com/learn-co-curriculum/phase-1-arrays-objects-what-is-really-going-on)](https://github.com/learn-co-curriculum/phase-1-arrays-objects-what-is-really-going-on) [(https://github.com/learn-co-curriculum/phase-1-arrays-objects-what-is-really-going-on/issues/new)](https://github.com/learn-co-curriculum/phase-1-arrays-objects-what-is-really-going-on/issues/new)

Learning Goals

- Recognize Arrays are Objects
- Recognize that many other things in JavaScript are Objects
- Take a deeper look at Objects
- Introduce `this`
- Introduce Prototypal Inheritance

Introduction

So far, we've seen that both Arrays and Objects can store things inside them, including *other* Arrays and Objects. We think this is pretty cool! You can use data to represent all sorts of things using nested data structures.

We'll soon see, however, that there is more going on. In this lesson, we're going to briefly explore what's really going on with Arrays and Objects behind the scenes.

> **Note:** Before we dive in too deep — some of the topics we will touch on in this lesson will be covered in more depth later on in this course. Do not feel that you need to fully understand concepts like context and prototypes. As you've already proven, data structures can be useful to us, even if we haven't fully understood them.

Arrays are... Objects in JavaScript?

If you recall from the previous lessons on functions, in JavaScript, functions are considered *first-class*. This means that, like data values, they can be used as arguments in other functions and assigned to variables. It also means you can store functions *in* Arrays and Objects. For example:

```
const phrases = {
  greeting: "Hello there!",
  time: () => {
    const currentTime = new Date();
    return `The time is ${currentTime.getHours()}:${currentTime.getMinutes()}`;
  }
}

phrases.greeting;
// => "Hello there!"
phrases.time();
// => "The time is 16:51" (or whatever time it is currently on a 24-hour clock)
```

Here, we've stored a function in an Object, and then called that function with `phrases.time()` . Let's break that down — we first call the `phrases` object. This is followed by a dot, `.` , then the key `time` . This key points to a value — a function expression. Adding parentheses, `()` , executes that function expression.

Now, hold on a moment — we've seen this dot syntax before, but with Arrays:

```
const listOfGoodDogs = ["Nola", "Spinach", "Diego"];

listOfGoodDogs.map((dog) => console.log(dog));
// LOG: Nola
// LOG: Spinach
// LOG: Diego
```

Here, we've called `map` on our array, `listOfGoodDogs` , and passed in a callback function to log each element in the Array. As with `time` in the previous example, `map` is acting like an Object key pointing to a function expression.

Why does this work? Well... it is because Arrays *are* Objects in JavaScript. Lots of things are Objects, actually. Notice in the two previous examples, we used the dot syntax for other things. In the first code snippet, we created a `const` , `currentTime` , assigned `new Date()` as its value, then called `getHours()` and `getMinutes()` on it. In the second code snippet, we called `log()` as part of `console` .

These are all JavaScript Objects — **[Arrays](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)** and other things like **[`Date`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)** are Objects... even ***[Strings](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)*** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)** are Objects, which is why we can do things like `"hello".slice(1)` . Functions... are also Objects in JavaScript if things weren't confusing enough already.

As it turns out, Objects are a bit more complex than we originally presented!

A Deeper Look at Objects

Before we continue, we want to be clear in the language we use going forward — so far we've talked about key/value pairs in general, but they're actually referred to as different things depending on what they store. Key/value pairs like `greeting` and `time` are also referred to as *properties* of an Object. Properties that store a function expression as a value, like `time` , are referred to as *methods* of the object. The `phrases` object we've defined, then, has two properties, one of which is a method.

We've gotten used to creating objects using the object literal notation, using curly braces to wrap comma separated properties:

```
const phrases = {
  greeting: "Hello there!",
  time: () => {
    const currentTime = new Date();
    return `The time is ${currentTime.getHours()}:${currentTime.getMinutes()}`;
  }
}
```

This way of creating Objects is often preferred due to its simplicity, but there are other ways we can create Objects. Say, for example, that we want to be able to create multiple Objects that all share some properties. Rather than type out all the properties each time, we can use a *Constructor function*.

Creating an Object Using the Constructor Function

We mentioned earlier that functions are Objects. The easiest way to demonstrate this is to create an object using a function. We can recreate our `phrases` object using what is called a 'Constructor' function:

```
function PhraseObjectConstructor(name) {
  this.greeting = `Hello there ${name}!`;
  this.time = () => {
    const currentTime = new Date();
    return `The time is ${currentTime.getHours()}:${currentTime.getMinutes()}`;
  };
}

const phrases = new PhraseObjectConstructor("Harold");


phrases.greeting;
// => "Hello there Harold!"
phrases.time();
// => "The time is 17:30"
```

We can see here that the code above results in a `phrases` object that behaves like the previous examples, with `greeting` and `time` properties. You probably notice some things that are unfamiliar, though.

Note that instead of using key/value pairs to set properties, we've used something else — `this` followed by the dot notation we've seen. We will go into greater depth on `this` and context later. For now, take note that in our example, `this` seems to be written like it is an Object itself; the properties we're assigning, `greeting` and `time` , are part of `this` .

Another noticeable difference is that `PhraseObjectConstructor()` does not *return* anything explicitly (the only `return` is inside the `time` method). However, when we run `new PhraseObjectConstructor("Harold")` , we do assign *something* to the `phrases` variable — *an Object*.

> *One more note:* Constructor functions start with a capital letter, by convention. This capitalization is a hint to let other programmers know that the function should be used as a constructor function, and to add the `new` keyword before calling the function.

The essential bit in this puzzle is **`new`** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new)** . Adding `new` before `PhraseObjectConstructor("Harold")` tells JavaScript to do a couple of things:

- It creates a basic Object (which gets assigned to the `phrases` variable).

- It binds `this` to the newly created Object. The properties defined in the function now belong to *this* new Object.
- It adds a new property, `__proto__` to the Object.

The first action is something we're familiar with, less so the other two. We'll discuss both then check out an example of why this behavior is useful.

A Brief Intro to `this`

`this` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) is a reserved word in JavaScript that returns the *context* it is in. The value of `this` depends on where and how it is used. Consider the following plain object:

```
const example = {
  name: "Henry",
  test: function() {
    return this;
  }
}

example.test();
// => {name: "Henry", test: f}
```

If you paste the above into your browser console and run `example.test()`, you will get the `example` object in return!

You may notice we're not using an arrow function here. If you replace `test` with an arrow function, you'll get a different value for `this`. The reason is beyond the scope of this lesson and is related to how context is determined in arrow functions.

`this` can be very useful since we can use it to reference objects from inside themselves.

```
const example = {
  name: "Henry",
  sayName: function() {
    return `My name is ${this.name}`;
  }
}

example.sayName();
// => "My name is Henry"
```

Going back to `new`, when we call `new PhraseObjectConstructor("Harold")`, `this` gets bound to the newly created object, turning `this.greeting` and `this.time` into properties for that object.

A Brief Intro to Prototypal Inheritance

We mentioned that when using `new`, a property `__proto__` is added to the newly created object. `__proto__` refers to an Object's **prototype (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes)**. Every JavaScript Object has a prototype property, though it isn't typically displayed when logging.

The prototype contains *inherited* properties, often methods. When we use a constructor function to create objects, the created object will inherit prototype properties from the constructor function (remember that it too is an Object). The constructor function has a prototype that *it* inherited, as well. In this way, some shared properties are able to be 'passed down' from Object to Object. This is known as a prototype *chain*. Properties of an Object that are in the prototype can be accessed using the `__proto__` property of an individual object.

Remember when we mentioned that Arrays are a *type* of Object and that there are many Objects in JavaScript? Once we create an array, we can access methods on that array to do things.

```
const exampleArray = [1, 2, 3];

exampleArray.pop();
// => 3
exampleArray;
// => [1, 2]
```

Methods like `pop()` (and `push()`, `shift()`, `unshift()`, etc...) are available on every Array we create because **these methods exist in the prototype shared by all Arrays**. We can actually see them if we use `exampleArray.__proto__`.

```
exampleArray.__proto__
// => {
// concat: f,
// constructor: f,
// ...
// ...
// pop: f,
// push: f,
// ...
// ...
// }
```

When we call `new PhraseObjectConstructor()` , a `PhraseObjectConstructor` prototype is passed to every object created. This prototype contains its own `__proto__` property, which points to the basic Object prototype that the `PhraseObjectConstructor` function inherited from.

```
Object -> PhraseObjectConstructor -> individual object
```

> **Note:** Remember, do not be discouraged if you find these concepts confusing. They are most definitely confusing and will remain that way for a bit, but that is okay. As you progress through the JavaScript content, you'll see more examples of `this` and prototypes.

Conclusion — The Power of Objects

Let's review what we've found out so far about Objects.

- We know they can contain properties
- We know `this` can be used in an object to reference itself
- We know Objects inherit shared properties from other Objects via the prototype chain
- We know many things in JavaScript are actually Objects
- There are multiple ways to create Objects

You may occasionally find programmers debating online as to whether or not JavaScript is an object-oriented language. Some resources will refer to JavaScript as having 'object-oriented capabilities' but not as 'object-oriented.' This is technically true, as JavaScript doesn't strictly adhere to some specific design principles related to object-orientation. However, we'll soon see that you can absolutely use JavaScript as you would use other object-oriented languages.

One core concept of object-orientation is the ability to create object 'classes.' A class can be thought of as a template; a blueprint we can use to create something from. In object-orientation, the things we create are typically referred to as 'instances.' Instances are individual copies of a class that can each carry unique information, but contain shared properties that were defined on the class.

Does this seem familiar? Sounds very similar to what we've discussed regarding constructor functions and prototypal inheritance. When we create a constructor function, we are essentially creating a template that can be used to generate new, individual objects.

```
function PhraseObjectConstructor(name) {
  this.greeting = `Hello there ${name}!`;
  this.time = () => {
    const currentTime = new Date();
    return `The time is ${currentTime.getHours()}:${currentTime.getMinutes()}`;
  };
}

const phrases1 = new PhraseObjectConstructor("Harold");
const phrases2 = new PhraseObjectConstructor("Hank");

phrases1.greeting;
// => "Hello there Harold!"
phrases2.greeting;
// => "Hello there Hank!"
```

These objects can store unique information in their properties, but share a similar structure and have both inherited the constructor function's prototype.

With this knowledge, we encourage you take a look back at some of the JavaScript you've used so far. Did you know you can create new Arrays with `new Array()` ? Can you guess what is happening when this command is run? What about other examples we've seen? `new Date()` is an interesting example — it *returns a string* when used, but it **also** can be used to create a `Date` object with unique properties like `getHours` and `getMinutes` .

Things may still seem mysterious, but keep these ideas in mind as you move through the remaining content. You'll see these concepts appear again, but they will hopefully not be so unfamiliar!

Resources

- `new` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new)
- **Object Prototypes** (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes)
- `this` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this)