Lesson Weekend

Intermediate JavaScript (/intermediate-javascript)
/ Object-Oriented JavaScript (/intermediate-javascript/object-oriented-javascript)
/ JavaScript Objects

Text Cheat sheet

JavaScript is an object-oriented programming language. We previously learned that JavaScript implicitly turns (some) primitives like strings, numbers, and booleans into objects. Well, arrays and even the functions we write are also objects in JavaScript. In this lesson, we will explore what it means to be an object in JavaScript.

We've worked a lot with built-in objects already, especially when working with Web APIs, which are the structures and tools that web browsers provide to developers so they can create interactive programs. At this point, we are well-versed on how to access object properties and methods. But we haven't yet created our own custom objects, and that's just what we'll learn how to do in this lesson! Given that we know a fair amount about objects, some of the information in this lesson will be a review and some of it will be new.

Code along with this lesson by adding the example objects to the DevTools console.

JavaScript Objects

We use objects when we want a variable to store much more information about the "thing" the variable represents. For example, if *you* were a variable and we wanted to store information about you as a student at Epicodus (your name, your courses, your track, your enrollment status) in the single variable called `epicodusStudent`, we'd need more than a single string or a single array — we'd need an object!

We know an object to be a collection of properties. We also understand that an object is a container for related but separate data. Well, in technical terminology, objects are containers that **encapsulate** data. This means that all of the relevant data and functions for the thing that a variable represents (like `epicodusStudent`) are kept together in a "capsule", better known as an **object**, that can be created and manipulated in our programs **as a single unit**.

Here is an example of an `epicodusStudent` object in JavaScript:

```
let epicodusStudent = {
  firstName: "Charlie",
  lastName: "Bucket",
  year: 2022,
  track: "Ruby and  React",
  courses: ["Intro to Programming", "Intermediate JavaScrip
t", "Ruby and Rails", "React"],
  enrollmentStatus: true
};
```

Let's take a look at how this object is defined. We have our variable `epicodusStudent`. We assign it the value of an object by using the curly braces, `{ }`. This is called **object literal notation**. Even though we didn't know it, we use **literal notation** when we create strings by using quotes, `"this is a string"`, and arrays by using brackets, `[1, 2, 3]`. To learn more, check out this reference page on literals MDN (https://developer.mozilla.org/en-US/docs/Glossary/Literal).

Inside the curly braces are six properties for our `epicodusStudent` object: `firstName`, `lastName`, `year`, `track`, `courses`, and `enrollmentStatus`. Every property of a JavaScript object consists of a **key-value** pair:

- The **key** is the variable that describes the kind of information to be stored.
- The **value** is the specific value of the key.

So, in our example, the property for the first name has a key called `firstName` and a value of `"Charlie"`, and the property for the last name has a key `lastName`, with a value of `"Bucket"`, and so on with the remaining keys `year`, `track`, `courses`, and `enrollmentStatus`.

Each key and value is separated by a colon `:`, and key-value pairs are separated from each other with a comma `,`.

We could write our object like this and it would also work:

```
let epicodusStudent = { firstName: "Charlie", lastName: "Bucket", year: 2022, track: "Ruby and  React", courses: ["Intro to Programming", "Intermediate JavaScript", "Ruby and Rails", "React"], enrollmentStatus: true };
```

However, the formatting of the object with each property (or, key-value pair) indented two spaces on a separate line is a convention used when writing JavaScript objects to make it easy to see each property. Imagine an object with hundreds of properties written on the same line. It would be a bit of a challenge to sort out the details.

Though quotes are not needed in this context, property keys are always a JavaScript string that starts with a letter. Note that sometimes you'll see built-in objects that have keys that are set to what seems like a number:

```
> const myObj = {
  0: "value",
  1: "value 2"
}
```

But, in reality these "numbers" are actually strings. This can be confusing, so be aware!

Property values can be any data type: string, number, boolean, array, object, or function. When the value of a property is a function, we call it a **method**.

Here is an object with one property and one method. Put it into the DevTools console. The method when called will make my cat "speak" by writing "Meow" to the console. (Previously, we have used `console.log` just for debugging, but we can use it for any message we'd like to see in the console.)

```
> let myCat = {
  name: "Kitty Poppins",
  speak: function() {
    console.log("Meow!");
  }
};
```

To keep it simple, you can think of properties as nouns and methods as verbs or actions.

So, once we have an object, how do we use it? What do we do with the `name` and `speak` method?

To access properties and methods on objects, we can use either **dot notation** or **bracket notation**.

```
> myCat.name;
"Kitty Poppins"
> myCat['name'];
"Kitty Poppins"
> myCat.speak();
Meow!
> myCat['speak']();
Meow!
```

Dot notation is easier to write and read, but **bracket notation allows us to use properties with special characters, or select properties using variables**. For example, if we use a number for a property name, we'll have to access it with bracket notation or else we'll get an error returned:

```
> let favColors = {
  1: "green",
  2: "fuscia"
};
> favColors.1
Uncaught SyntaxError: Unexpected number
> favColors["1"];
"green"
```

Let's create an empty new dog object in the DevTools console. We use the curly braces to signal JavaScript to create a new object.

```
> let dog = {};
undefined
> dog;
{}
```

The built-in JavaScript function that creates a new dog object returns the value of `undefined` but if we type `dog;` we can see that an empty object has been created for the `dog` variable.

Now, let's give our dog some properties using dot notation. Here our property values are a string and a number:

```
> dog.name = "Bark Twain";
"Bark Twain"
> dog.age = 5;
5
```

Now, let's add an array for a property value:

```
> dog.colors = ["brown","black","white"];
["brown","black","white"]
```

The value of a property comes with all of the functionality of its type. For example, we are able to use indexing on the `colors` array as we've done with other arrays.

```
> dog.colors[0];
"brown"
> dog.colors[1];
"black"
```

Or, we could call `String.prototype.toUpperCase()` on `dog.name`:

```
> dog.name.toUpperCase();
"BARK TWAIN"
```

We can use array methods on the `colors` property, like with `Array.prototype.push()` in the following example. Note, this method returns the new length of the array.

```
> dog.colors.push("gray");
4
> dog.colors;
["brown","black","white","gray"]
```

Number methods on the `age` property:

```
> dog.age;
5
> dog.age + 10;
15
```

We can also update any property by reassigning its value:

```
> dog.name = "Rex";
"Rex"
```

Let's add a method to our dog. This will be a property with a function as a value. In this case, we'll give our dog some `howl` functionality.

```
> dog.howl = function() {
  console.log("Aaaaaaaaaaaooooooooooooo!")
}
ƒ () {
  console.log("Aaaaaaaaaaaooooooooooooo!")
}
> dog.howl();
Aaaaaaaaaaaooooooooooooo!
```

With objects, we can use properties within other properties. What if we decided we wanted to calculate our dog's age in human years? Let's add another method to our `dog` object.

```
> dog.humanYears = function() {
  return this.age * 7
}
ƒ () {
  return this.age * 7
}
```

Notice that the `humanYears` function has a keyword of `this`. When `this` is used in an object's method, it always refers to the object on which the method is called. So, when we run `dog.humanYears()`, `this` will always refer to the object, `dog`. (`this` can also be used in other places, but it gets tricky depending on its context and we won't cover it in detail here.) We'll see more examples of this in coming lessons!

Now when we run `dog.humanYears()`, we get 35.

Using `const` to Declare Objects

You've probably noticed that we are using `let` instead of `const` to declare the objects above even when we aren't changing the object. This is intentional.

We can use `const` with objects — and it can still be very helpful for communicating our intentions to other developers. However, JavaScript doesn't do a good job making sure objects that are initialized with `const` are constant and don't change.

Let's take a look at an example in the DevTools console:

```
> const greetings = {};
> greetings.english = "hello";
> greetings.english;
"hello"
> greetings.english = "hey";
> greetings.english;
"hey"
```

We start by creating an object that is a constant. Then we add a property `english` to it. As you can see, there is no warning or error. Instead, the object is modified just as if we've used `let` instead.

It's very important to know about this gotcha. However, even though we can't use `const` to freeze the properties of an object, we can still use it to communicate about our code. Even though another developer (or us) can still modify an object that's declared with `const`, if we use `const`, we are letting everyone know that the object *shouldn't* be changed.

In this course, we'll generally create objects that will be modified, which is why we usually use `let`, not `const`. However, if you do create an object that should never be modified, make sure to use `const` instead.

Additional Practice

In the DevTools JavaScript console, practice creating objects of your own. Here is some guided practice to try.

Create an object that stores information about a flower including name, color, and height.

- Change the color of your flower using dot notation.
- Change the height of your flower using bracket notation.
- Add a property that indicates what kind of creatures help the flower with pollination. This includes bees, butterflies, and birds.
- Add one more creature to your list: humans.
- Write a method that allows the flower to grow. After the method is run, the height value should be increased.
- View all of the properties and methods for your flower object.
- Explore viewing, adding and updating more properties and methods on your flower object.
- Try using some of the string, number, and array methods you have used before on the properties that store these types of data.

Previous (/intermediate-javascript/object-oriented-javascript/homework-and-class-structure)
Next (/intermediate-javascript/object-oriented-javascript/literal-notation-versus-constructors)

Lesson 4 of 33
Last updated March 23, 2023

disable dark mode

Epicodus (http://www.epicodus.com)