Lesson Weekend

Introduction to Programming (/introduction-to-programming) / JavaScript and Web Browsers (/introduction-to-programming/javascript-and-web-browsers) / Another Look at JavaScript Objects

Text Cheat sheet

Let's take another look at JavaScript objects. Doing so will help us better understand JavaScript primitives and methods. This review will also help us understand how powerful objects are as a data structure, both in JavaScript and in the web browser tools we will begin to use soon, and prime us to use built-in objects. The goal of this lesson is to simply understand how important objects are.

Note that this lesson will show you how to create an object, but only as a demonstration of the structure of objects and their properties. You will never be required to create your own objects in your code until we get to the course section dedicated to object-oriented JavaScript.

Another Look at Objects

When we first talked about objects (https://www.learnhowtoprogram.com/introduction-to-programming/javascript-and-web-browsers/javascript-data-types)

we introduced these as containers for related data, data that can be *any* JavaScript data type.

> In JavaScript, we use objects to represent things and concepts — like animals, political parties, and computers. A JavaScript **object** is simply a container for related data. We can describe an object by listing its data and functionality, the same as describing a computer by listing its features and what it can do.
>
> More generally speaking, this definition of an object is true across computer programming languages. As we'll learn later in this section, our web browsers use objects to represent things like the browser window and the webpages that we create with HTML and CSS.
>
> JavaScript objects can hold multiple types of data and they can be assigned many different types of functionality. The data describes what an object is, and the functionality describes what an object can do. This is in contrast to JavaScript primitives which only represent one piece and type of data. Also, primitives don't *do* anything, meaning we can't give them actions to perform.

We also used a few examples to understand objects. Let's revisit the cat example. In this example we explained that a cat can be described by listing its features and what it can do. We learned that this information is exactly what we would include in the object representation of the cat: data about its features and functionality to perform actions.

> Let's consider another example object: a cat. We can describe what a cat is by itemizing different aspects about it. A cat has:
>
> - eyes and other organs
> - colors in its coat
> - a name
> - a personality
> - age
> - favorite activities
> - perhaps a family and friends?
> - how about enemies or prey?

A cat object's data can also describe what a cat can do:

> And what can a cat do? A cat can:
>
> - purr
> - scratch up the furniture
> - eat
> - meow
> - hunt
> - jump, and certainly more!

To describe a cat's colors, name, age, etc., we use JavaScript primitives: numbers, strings, booleans, and more. To describe what a cat can do, we use methods (because methods are a type of function that belongs to a specific data type, including objects).

Even though we have not seen a JavaScript object written in code, we should be able to tell that objects are really powerful tools in JavaScript that bundle related data in one package.

Comparing Primitives and Objects

It's helpful to compare JavaScript objects with JavaScript primitives. Notably, JavaScript primitives can only represent a single type and piece of data. JavaScript primitives can't do anything, and they can't contain multiple data types. A primitive can only represent the singular primitive type that it is, like a string, a number, or a boolean.

Conversely, JavaScript objects can hold many pieces of data of many different data types. We can also give objects functionality, called methods. Together, these pieces of data describe the object: what it is and what it can do.

JavaScript Object Syntax

To make the distinctions between JavaScript primitives and objects clear, let's look at a coded example of an object. Before we dive into it, remember that we'll be covering objects in depth in a dedicated course section. That means there's a lot to learn about objects, and right now, we're only covering the basics. The goal of showing you a coded object is to make the conceptual more concrete. Note that you do not have to try out the following examples in the browser DevTools console, but you can if you want to.

It's also helpful to have a reminder here that you don't need to know everything about how JavaScript works in order to use JavaScript. In this section, we won't create or use custom JavaScript objects, but we will be interacting with built-in objects a lot!

So, here's our cat, in object form, with some different data:

```
let cat = {
  name: "Waffles",
  age: 13,
  likesTunaFish: true,
  likesComputerProgramming: false,
  talk: function() {
    return "meow!";
  }
}
```

In the object above, our cat's name is "Waffles" and its age is 13. Waffles the cat likes tuna fish, but does not like to program computers. Waffles can also talk, saying "meow!".

Here's a look at object syntax in pseudocode:

```
// this is pseudocode!
let nameOfObjectType = {
  propertyName: value,
  secondPropertyName: otherValue,
  thirdPropertyName: yetAnotherValue
}
```

Let's use the pseudocode syntax above to break down the parts of our cat object. Compare it to the code snippet below. Here, we create a variable called `cat` and use the assignment operator `=` to set the value of the `cat` variable equal to an object. The object is denoted by the curly brackets `{ }`. All of the `cat` object's data is defined within the two curly brackets `{ }`. We're naming our object `cat`, because we want to create an object type that represents a cat.

```
let cat = {
  // all data about our object goes here.
}
```

The cat's data is defined through **properties** and **values**. A property is a variable that belongs to an object. We define a property and its value with this syntax: `propertyName: value`. Because of this, **JavaScript objects are defined as a collection of properties**. Let's look at the first property of our cat object to understand this new syntax:

```
name: "Waffles"
```

Here we have a property called `name` that's set to a value of `"Waffles"`, of the string type. This property of our cat object is describing our cat's name which is Waffles.

In our cat object we have 5 properties: `name`, `age`, `likesTunaFish`, `likeComputerProgramming`, and `talk`. All of these properties describe different features of our cat, and they are all assigned values of different data types:

```
name: "Waffles",                  // "name" property is a st
ring
age: 13,                          // "age" property is a num
ber
likesTunaFish: true,              // "likesTunaFish" propert
y is a boolean
likesComputerProgramming: false, // "likesComputerProgrammi
ng" property is a boolean
talk: function() {                // "talk" property is a me
thod
  return "meow!";
}
```

Note `talk` is a method of the cat object, and not a function. The function declaration syntax `function() { }` is used to create methods. We know `talk` is a method in this context because it belongs to the `cat` object, and doesn't exist outside of it. Remember, methods always belong to some data type, which includes objects. Don't worry if this distinction is not clear — we will practice with this more soon!

Here's how we interact with our cat object:

```
> cat.name;
"Waffles"
> cat.likesTunaFish;
true
> cat.talk();
"meow!"
```

When we write `cat.name`, we're telling JavaScript to look in the `cat` object and access the `name` property and return the value `"Waffles"`. When we write `cat.name`, we're using **dot notation** to access the `name` property of the `cat` object, by including a dot `.` after the name of the object.

For `cat.talk()`, this can be a bit more confusing for folks who are new to JavaScript. What we're doing here is asking JavaScript to look in the `cat` object and access the `talk` property; since the value of `talk` is set to a function that means we can call on it. So, we add parentheses to call `cat.talk()` and execute the functionality. Note, `talk` is a method of the cat object, and not a function. We can't call `talk()` on its own, we always have to call it on the cat object, like `cat.talk()`. This is just like calling a string method on a string:

```
> "Hello, Waffles!".toUpperCase();
"HELLO, WAFFLES!"
```

If anything we've covered so far doesn't make complete sense, that's okay. The goal of reviewing JavaScript object syntax is to make our conceptual discussion about JavaScript objects more concrete. We don't actually need to understand how to use objects right now. However, you should understand these takeaways:

- Objects represent things and concepts, like cats, grocery bags, and shopping carts.
- Objects bundle separate pieces of data into one package.
- Data is separated and saved into properties that have values; this is why objects are considered to be collections of properties.
- Objects can hold many pieces of data of any data type.
- Objects describe both what something is and what it can do.

The Benefits of Using Objects

It's worth considering how else we could represent our cat and all of its data. Given what we know about JavaScript, the best option for us is a string:

```
"The cat's name is 'Waffles'. The cat's age is 13. The cat likes tuna fish. The cat does not like to program computers. The cat says 'meow!'."
```

However, when we use a string to describe all of this data, we are more limited. For example, if we needed to change Waffle's age because of a recent birthday, we'd have to rewrite our entire string:

```
"The cat's name is 'Waffles'. The cat's age is 14. The cat likes tuna fish. The cat does not like to program computers. The cat says 'meow!'."
```

When we itemize our cat into many different data types in one cat object, we're packaging related data in one bundle, but we're maintaining the flexibility to work with different data types. This is the power of objects in action.

If we want to update Waffles's age, we could do the following (if you try this, you need to input the original cat object in your console):

```
> cat.age += 1;
> cat.age;
14
```

Or if we want to make a string that introduces Waffles, we could do this:

```
> "The cat's name is '" + cat.name + "'. The cat's age is "
+ cat.age.toString() + ".";
```

JavaScript Turns (Some) Primitives into Objects

Because objects are so powerful, JavaScript actually turns some primitives into objects. Why? In order to give them more complex functionality. Let's read a section from the "Primitive" reference page on MDN (https://developer.mozilla.org/en-US/docs/Glossary/Primitive#primitive_wrapper_objects_in_javascript) that describes this:

> Except for `null` and `undefined`, all primitive values have object equivalents that wrap around the primitive values:
>
> - `String` for the string primitive.
> - `Number` for the number primitive.
> - `BigInt` for the bigint primitive.
> - `Boolean` for the boolean primitive.
> - `Symbol` for the symbol primitive.
>
> The wrapper's valueOf() method returns the primitive value.

To understand this, let's use the string primitive as an example. What this means is that every time we write a string like this:

```
> "hello world"
"hello world"
```

Behind-the-scenes, JavaScript turns the string into an object, and not just any object, but a `String` object that represents strings only. There's no indication of JavaScript turning primitives into objects, it just happens. Also, the primitive value of the string doesn't disappear, it becomes a part of the object! If we want to access the primitive value of the string, we can use the `valueOf` method:

```
> "hello world".valueOf();
"hello world"
```

However, this method is not particularly useful to us. JavaScript does use that method internally, but that's not important to know about now.

Do you remember when we said that JavaScript primitives can't have functionality like methods? This means that primitives can't *do* anything, they just exist as the primitive they represent, like the number `4` or the boolean `false`. Thanks to JavaScript turning (some) primitives into objects, JavaScript can assign them functionality like built-in methods. This is why we can call methods on strings:

```
> "Methods belong to objects".toUpperCase();
"METHODS BELONG TO OBJECTS"
```

Now, do you remember when we discussed that methods always belong to some data type? By this we mean that `String.prototype.toUpperCase()` belongs to strings only, and `Number.prototype.toFixed()` belongs to numbers only. Well it's more accurate to say that **methods always belong to some object type**. This is because primitives can't have functionality like methods. MDN really drives this home on its glossary page for Primitives (https://developer.mozilla.org/en-US/docs/Glossary/Primitive):

> In JavaScript, a primitive (primitive value, primitive data type) is data that is not an object and has no methods.

Documentation on Built-In Objects

MDN has excellent documentation on all built-in JavaScript objects. In the coming sections, we'll learn about new objects, including their features and methods.

For now, take a look at the MDN page on Standard Built-In Objects:

- **Standard built-in objects (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)**

MDN has organized these objects into different categories. Don't worry about understanding everything that's on the page, but scroll through the page and find these objects:

- String — categorized as "text processing"
- Number — categorized as "numbers and dates"
- Boolean — categorized as "fundamental objects"

Summary

In this lesson, we learned that objects are more complex and powerful than primitives. Like we saw in the cat object example:

- Objects are containers that hold one or more pieces of related data.
- The data is saved in properties.
- The data can be of any data type.

Conversely, primitives can only represent a singular piece of data and type.

We also learned that primitives don't have methods. Instead, methods belong to objects. We always have to call the method on the object it belongs to. In other words, when a function belongs to an object, we call it a method.

JavaScript strings, numbers, and booleans are implicitly turned into objects when we use them. JavaScript does this so that it can give these data types more complex and built-in functionality, like methods. This is also why we are able to call built-in methods on strings, numbers, and booleans — it's because they are primitives that have been turned into objects!

Going forward, we won't worry about distinguishing between the primitive or the object of data types like strings, booleans, and numbers. It's not important. However, knowing the difference between primitives and objects helps us better understand how JavaScript is structured and how to use MDN documentation.

Finally, it's worth reiterating that we don't need to remember how to create an object! Instead we want to know what objects are and what they can do. Soon, we'll learn about more built-in objects and how to make use of them in our code.

Previous (/introduction-to-programming/javascript-and-web-browsers/data-types-detection-conversion-and-review)
Next (/introduction-to-programming/javascript-and-web-browsers/review-of-javascript-conventions)

Lesson 21 of 75
Last updated more than 3 months ago.