Objects

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

Learning Goals

- Identify JavaScript `Object` s
- Access a value stored in an `Object`
- Learn about JavaScript's Object methods

Introduction

While `Array` s are great for representing simple, ordered data sets, they're generally not so great at modeling a more complex structure. For that, we need `Object` s. In this lesson, we'll begin to learn about `Object` s, specifically, what they are, how to create them, and how to access values stored in them. In the next lesson, we'll learn how to modify `Object` s, both destructively and nondestructively.

Be sure to follow along with the examples in this lesson in **replit** [(https://replit.com/languages/javascript)](https://replit.com/languages/javascript).

> **ASIDE**: You may have heard about or be familiar with the concept of *object-oriented programming*. Un-helpfully JavaScript called this thing with curly braces (`{}`) an `Object` , but it is not related to object orientation. There was initially no thought that JavaScript would ever need to accommodate object-oriented programming but as it grew in popularity, the ability to use object orientation was added to the language. A JavaScript `Object` , however, is a *data structure* consisting of `key` s and `value` s, similar to Ruby's `Hash` , Python's `Dictionary` or C-like languages' `struct` (ure). It is important not to confuse the two.

Identify JavaScript Objects

Let's think about how we could represent a company's address in JavaScript. Addresses are made up of words and numbers, so at first it might make sense to store the address as a string:

```
const address = "11 Broadway, 2nd Floor, New York, NY 10004";
```

That looks decent enough, but what happens if the company moves to a different floor in the same building? We just need to modify one piece of the address, but with a string we'd have to involve some pretty complicated find-and-replace pattern matching or replace the entire thing. Instead, let's throw the different pieces of the address into an `Array` :

```
const address = ["11 Broadway", "2nd Floor", "New York", "NY", "10004"];
```

Now, we can just grab the small piece that we want to update and leave the rest as is:

```
address[1] = "3rd Floor";

address;
//=> ["11 Broadway", "3rd Floor", "New York", "NY", "10004"]
```

This seems like a better solution, but it still has its drawbacks. Namely, `address[1]` is a **terrible** way to refer to the second line of an address. What if there is no second line, e.g., `['11 Broadway', 'New York', 'NY', '10004']` ? Then `address[1]` will contain the city name instead of the floor number.

We could standardize it, putting an empty string in `address[1]` if there's no second line in the address, but it's still poorly named. `address[1]` offers very little insight into what data we should expect to find in there. It's a part of an address, sure, but which part?

To get around this, we could store the individual pieces of the address in separate, appropriately-named variables:

```
const street1 = "11 Broadway";
const street2 = "2nd Floor";
const city = "New York";
const state = "NY";
const zipCode = "10004";
```

That's solved one issue but reintroduced the same problem we tackled in the lesson on `Array` s: storing pieces of related data in a bunch of unrelated variables is not a great idea! If only there were a best-of-both-worlds solution — a way to store all of our address information in a single data structure while also maintaining a descriptive naming scheme. The data structure we're after here is the ***`Object`***.

What Is an Object?

Like `Array` s, JavaScript `Object` s are collections of data. They consist of a list of *properties* (*key-value pairs*) bounded by curly braces (`{ }`). The properties can point to values of any data type — even other `Object` s.

We can have empty `Object` s:

```
const obj = {};
```

Or `Object` s with a single property:

```
const obj = { key: value };
```

When we have to represent multiple properties in the same `Object` (which is most of the time), we use commas to separate them:

```
const obj = {
  key1: value1,
  key2: value2,
};
```

We can also have nested `Object` s, in which the values associated with one or more of the keys is another `Object` :

```
const obj = {
  key1: value1,
  key2: {
    innerKey1: innerValue1,
    innerKey2: innerValue2,
  },
};
```

There is no limit to how deeply nested our `Object` s can be.

For a real example, let's define our address as an `Object` :

```
const address = {
  street: {
    line1: "11 Broadway",
    line2: "2nd Floor",
  },
  city: "New York",
  state: "NY",
  zipCode: "10004",
};
```

Here we're defining `address` using `Object` literal syntax: literally typing out the `Object` inside the `{}` . Our address `Object` has four keys: street, city, state, and zipCode. The first key points to another `Object` which itself has two keys: line1 and line2. Note that there's a comma between each key-value pair in both the top-level `Object` and the nested `Object` . Practice creating an object in the REPL; then try leaving out a comma to see what happens.

Multiple properties can have the same value:

```
const meals = {
  breakfast: "Avocado toast",
  lunch: "Avocado toast",
  dinner: "Avocado toast",
};

meals.breakfast;
// => "Avocado toast"

meals.dinner;
// => "Avocado toast"
```

But keys must be unique. If the same key is used for multiple properties, only the final value will be retained. The rest will be overwritten:

```
const meals = {
  breakfast: "Avocado toast",
  breakfast: "Oatmeal",
  breakfast: "Scrambled eggs",
};

meals;
// => { breakfast: "Scrambled eggs" }
```

The real data in an `Object` is stored in the *value* half of the key-value pairings. The *key* is what lets us access that value. In the same way we use *identifiers* to name variables and functions, inside an `Object` we assign each value a key. We can then refer to that key and the JavaScript engine knows exactly which value we're trying to access.

Access a Value Stored in an Object

We access an `Object` stored in a variable in the same way we access any variable's value: by typing the variable name. Then, to access one of the values *inside* our `Object` , we add a reference to the key associated with the value we want; we can do that using either *dot notation* or *bracket notation*.

Dot Notation

With *dot notation*, we use the *member access operator* (a single period) to access values in an `Object` . For example, we can grab the individual pieces of our address, above, as follows:

```
address.street;
//=> { line1: "11 Broadway", line2: "2nd Floor" }

address.city;
//=> "New York"

address.state;
//=> "NY"
```

```
address.zipCode;
//=> "10004"
```

Then to access a value inside `address.street` , we simply append the inner key, again using dot notation:

```
address.street.line1;
//=> "11 Broadway"

address.street.line2;
//=> "2nd Floor"
```

> **Note**: You might initially think we should use `address.line1` to access the value associated with the `line1` key, but `address` and `address.street` are *separate* `Object` s. `address.street` is the *identifier* for the nested `Object` — the one that contains the `line1` key — and we can use dot notation on that just the same as on the top-level `Object` , `address` . Experiment in the REPL to make sure you understand how it works.

Dot notation is fantastic for readability, as we can just reference the bare key name (e.g., `street` or `zipCode`). Because of this simple syntax, it should be your go-to strategy for accessing the properties of an `Object` .

> **NOTE**: Most people just call it *dot notation* or the *dot operator*, so don't worry too much about remembering the term *member access operator*.

Accessing Nonexistent Properties

If we try to access the `country` property of our `address` `Object` , what will happen?

```
address.country;
//=> undefined
```

It returns `undefined` because there is no matching key on the `Object` . JavaScript is too nice to throw an error, so it lets us down gently. Keep one thing in mind, though: if you're seeing `undefined` when trying to access an `Object` 's properties, you should recheck which properties exist on the `Object` (along with your spelling and capitalization)!

Bracket Notation

With *bracket notation*, we use the *computed member access operator*, which, recall from the lesson on `Array` s, is a pair of square brackets (`[]`). To access the same properties as above, we need to represent them as strings inside the operator:

```
address["street"];
//=> { line1: "11 Broadway", line2: "2nd Floor" }

address["street"]["line1"];
//=> "11 Broadway"

address["street"]["line2"];
//=> "2nd Floor"

address["city"];
//=> "New York"

address["state"];
//=> "NY"

address["zipCode"];
//=> "10004"
```

Bracket notation is a bit harder to read than dot notation, so we always default to the latter. However, there are two main situations in which we need to use bracket notation.

Nonstandard Keys

If (for whatever reason) you need to use a nonstandard string as the key in an `Object` , you'll only be able to access the properties with bracket notation. For example, this is a valid `Object` :

```
const wildKeys = {
  "Cash rules everything around me.": "Wu",
  "C.R.E.A.M.": "Tang",
  "Get the money.": "For",
  "$ $ bill, y'all!": "Ever",
};
```

It's impossible to access those properties with dot notation:

```
wildKeys.'Cash rules everything around me.';
// ERROR: Uncaught SyntaxError: Unexpected string
```

But bracket notation works just fine:

```
wildKeys["$ $ bill, y'all!"];
//=> "Ever"
```

In order to access a property via dot notation, **the key must follow the same naming rules applied to variables and functions**. It's also important to note that under the hood **all keys are strings**. Don't waste too much time worrying whether a key is accessible via dot notation. If you follow these rules when naming your keys, everything will work out:

- camelCaseEverything
- Start the key with a lowercase letter
- No spaces or punctuation

If you follow those three rules, you'll be able to access all of an `Object` 's properties via bracket notation **or** dot notation.

> **Top Tip**: Always name your `Object` 's keys according to the above three rules. Keeping everything standardized is good, and being able to access properties via dot notation makes the code much more readable.

Accessing Properties Dynamically

The other situation in which bracket notation is required is if we want to dynamically access properties (i.e., using variables rather than literal identifiers). The reason we need to enclose the key inside quotes when we use the literal key is because, when we **don't** use the quotes, JavaScript will interpret what's inside the brackets as a variable.

From the lesson on `Array` s, remember why we call it the *computed member access operator*: we can place any expression inside the brackets and JavaScript will *compute* its value to figure out which property to access. For example, we can access the `zipCode` property from our `address` `Object` like so:

```
address["zip" + "Code"];
//=> "10004"
```

Pretty neat, but the real strength of bracket notation is its ability to compute the value of variables on the fly. For example:

```
const meals = {
  breakfast: "Oatmeal",
  lunch: "Caesar salad",
  dinner: "Chimichangas",
};

let mealName = "lunch";

meals[mealName];
//=> "Caesar salad"
```

By placing `mealName` in the square brackets, we're telling the JavaScript engine it needs to *interpret* the value inside those brackets. It evaluates `mealName` , resolves it to `'lunch'` , and returns "Caesar salad". Note that we didn't enclose the key in quotes: the keys themselves are strings, but `mealName` is a variable *containing* a string. If we try to use the `mealName` variable with dot notation instead, it doesn't work:

```
mealName = "dinner";
//=> "dinner"
```

```
meals.mealName;
//=> undefined
```

With dot notation, JavaScript doesn't treat `mealName` as a variable — instead it checks whether a property exists with the literal key `mealName`, only finds properties named `breakfast`, `lunch`, and `dinner`, and so returns `undefined`. Essentially, dot notation is for when you know the exact name of the property in advance, and bracket notation is for when you need to compute it when the program runs.

The need for bracket notation doesn't stop at dynamically accessing properties on an already-created `Object`. We can also use bracket notation to dynamically set properties *during the creation of a new* `Object`. For example:

```
const morningMeal = "breakfast";
const middayMeal = "lunch";
const eveningMeal = "dinner";

const meals = {
  [morningMeal]: "French toast",
  [middayMeal]: "Personal pizza",
  [eveningMeal]: "Fish and chips",
};

meals;
//=> { breakfast: "French toast", lunch: "Personal pizza", dinner: "Fish and chips" }
```

Once again, by wrapping the variable names in square brackets, we're letting JavaScript know that it needs to interpret the contents. Let's try doing the same thing without the square brackets:

```
const morningMeal = "breakfast";
const middayMeal = "lunch";
const eveningMeal = "dinner";

const meals = {
  morningMeal: "French toast",
  middayMeal: "Personal pizza",
  eveningMeal: "Fish and chips",
};

meals;
//=> { morningMeal: "French toast", middayMeal: "Personal pizza", eveningMeal: "Fish and chips" }
```

Without the square brackets, JavaScript treated each key as a literal identifier instead of a variable. Bracket notation — the *computed member access operator* — once again shows its powers of computation!

Bracket notation will really come in handy when we start iterating over `Object` s and programmatically accessing and assigning properties.

JavaScript's Object Methods

JavaScript includes a number of built-in **static `Object` methods (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static)** (*static* means that these methods are called on the `Object` class itself, rather than on an instance of an `Object`). We will talk briefly about a couple of them here and go into detail about one more in the next lesson.

`Object.keys()`

We can get a list of the top-level keys in an `Object` by using the **`Object.keys()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys)** static method. We do that by calling `Object.keys()` and passing the `Object` instance as an argument. The return value is an `Array` containing all of the keys at the *top level* of the `Object` instance.

```
const wednesdayMenu = {
  cheesePlate: {
    soft: "Brie",
    semiSoft: "Fontina",
    hard: "Provolone",
  },
  fries: "Sweet potato",
  salad: "Southwestern",
};
```

```
Object.keys(wednesdayMenu);
//=> ["cheesePlate", "fries", "salad"]
```

Notice that it didn't pick up the keys in the nested `cheesePlate` `Object` — just the keys from the properties declared at the top level within `wednesdayMenu`. How do you think we could use `Object.keys()` to get a list of the keys inside the nested `Object`? Try it out in the REPL.

> **NOTE**: The sequence in which keys are ordered in the returned `Array` is not consistent across browsers and should not be relied upon. All of the `Object`'s keys will be in the `Array`, but you can't count on `keyA` always being at index `0` of the `Array` and `keyB` always being at index `1`.

`Object.values()`

The `Object.values()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values) static method behaves similarly to `Object.keys()` but, as you might expect, returns an array containing values rather than keys. Try this out in the REPL as well.

Conclusion

In this lesson, we identified what an `Object` is and how it gives us a better way to keep track of more complicated sets of related data. We also learned how to access values stored in an `Object` using dot notation and bracket notation, as well as when to use each. In the next lesson, we'll learn how to modify `Object` s. We'll also explore the relationship between `Object` s and `Array` s.

Creating and interacting with `Object` s is an important skill in JavaScript programming. Before moving on, be sure to use **replit** (https://replit.com/languages/javascript) to practice creating `Object` s (including nested `Object` s) and accessing properties.

Resources

- MDN
  - **Object basics** (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics)