Introduction to JavaScript Objects

By the end of this lesson, you'll have the knowledge and skills to produce your own JavaScript objects.

Throughout this lesson, you will learn how to:

* Represent real-world objects in JavaScript
* Access object properties
* Access object methods
* Create object getter and setter methods

These concepts will enable you to bundle related data and methods into one package that models the real world.

Let's get started!

Objects

JavaScript *objects* are containers that can store data and functions. The data we store in an object is not ordered — we can only access it by calling its associated *key*.

You can create an object with *key-value* pairs using the following syntax:

let restaurant = {

name: 'Italian Bistro',

seatingCapacity: 120,

hasDineInSpecial: true,

entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine Pesto']

};

Let's consider the above example one step at a time:

1. let restaurant creates a variable named restaurant that stores the object.
2. We create the object between curly braces: {}.
3. name, seatingCapacity, hasDineInSpecial, and entrees are all keys.
4. We separate each key from its corresponding value by a colon (:).
5. The value is to the right of the colon. For example, seatingCapacity's value is 120.
6. Every key-value pair is separated by a comma ,.

An objects keys point to values that can be any data type, including other objects.

# Accessing Object Properties I

Now that we have data in an object, we need a way to access the values. The most common way to access a key's value is to use dot notation.

Look at how entrees is accessed from the restaurant object in last line of the example below:

let restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'] }; console.log(restaurant.entrees);

The output would be:

['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto']

To access the properties within an object, we connect the value's name to the key name with a period .. By using console.log() to print restaurant.entrees, we can see the value of the key entrees.

# Accessing Object Properties II

Another way to access a key's value is with bracket notation.

Just like with dot notation, we can use opening ([) and closing (]) brackets to access a key. The syntax looks like this:

let restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'] }; console.log(restaurant['entrees']);

The output would, again, be:

['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto']

On the last line of the program, we used restaurant['entrees']. This syntax uses bracket notation to select the value at the entrees key from within the restaurantobject.

Notice that bracket notation outputs the same value as dot notation. You may see either of these techniques used in JavaScript programs, so it's good to be familiar with both.

Accessing Object Properties III

One advantage that bracket notation has over dot notation is that you can use variables inside the brackets to select the keys of an object.

For example, a restaurant may have different specials based on the time of day. We could put each special in the restaurantSpecialsobject, then select the one we need later in our program based on the current time.

let meal = 'none'; let time = 12; // We'll use military time for this example, counting hours 0-23. const restaurantSpecials = { breakfast: 'The breakfast special is 20% off freshly squeezed orange juice', lunch: 'The lunch special is 10% off appetizers', none: 'There are no specials currently' }; if (time < 11) { // 11 am meal = 'breakfast'; } else if (time < 17) { // 5 pm meal = 'lunch'; } console.log(restaurantSpecials[meal]);

The output would be:

The lunch special is 10% off appetizers

1. The restaurantSpecials object has three key-value pairs for different specials throughout the day: breakfast, lunch, and none.
2. The if/else statement sets the mealvariable to 'breakfast' or 'lunch'based on the the time. For purposes of this example, we can imagine the timevariable getting updated every hour.
3. On the last line, we wrote restaurantSpecials[meal]. The mealvariable is not a key in the restaurantSpecials object. Because we are using bracket notation, JavaScript looks at the meal variable's value. In this case, meal is set to 'lunch' within the if/else statement because time is equal to 12. Since special equals 'lunch', writing restaurantSpecials[meal] is the same as writing restaurantSpecials['lunch']— the code outputs the lunch special.

Bracket notation is required to use variables to look up keys within an object. It's not possible to use variables like this with dot notation.

# Adding a Property

What if we want to add an item to our object after we've created it?

We can do this by assigning a value to a new key that's attached to our object. We attach a new key to our object using dot or bracket notation.

Objects are considered mutable, which means you can change them after they're created. Even if you save an object to a const variable, you can still add to and edit the key-value pairs inside of it without causing an error.

As an example, if we have this object:

const restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'] }

We can add appetizers and desserts keys by writing:

restaurant['appetizers'] = ['Fried Calamari', 'Bruschetta']; restaurant.desserts = ['Homemade Tiramisu', 'Cannoli'];

Since appetizers and desserts don't exist in the restaurant object, these lines will create and add the key-value pairs.

# Editing a Property

In the same way that we added a property to an object, we can modify a key's value.

In the last exercise, we added two properties to the restaurant object (appetizers and desserts).

We can replace the values assigned to appetizers and desserts with new arrays like this:

restaurant['appetizers'] = ['Fried Calamari', 'Bruschetta', 'Caprese Salad']; restaurant.desserts = ['Homemade Tiramisu', 'Canolli', 'Cheesecake'];

We used bracket and dot notations to specify which keys we wanted to select.

Then we set its value to a new array, this time with one more array item inside it.

# Methods

So far, we've paired keys with strings, numbers, booleans, and arrays. In this exercise, we'll show you how to pair keys with functions.

When objects have key-function pairs, we call the function a method. It looks like this:

const restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'], openRestaurant: () => { return 'Unlock the door, flip the open sign. We are open for business!'; }, closeRestaurant: () => { return 'Lock the door, flip the open sign. We are closed.' } }; console.log(restaurant.openRestaurant()); console.log(restaurant.closeRestaurant());

In the example above, we use arrow syntax to create two methods, .openRestaurant() and .closeRestaurant() in the restaurantobject. When we call these methods they return text, which we log to the console. The text in the console appears as follows:

Unlock the door, flip the open sign. We are open for business! Lock the door, flip the open sign. We are closed.

# Methods: ES6

In 2015, a new version of JavaScript (ES6) was introduced with additional methods and new syntax. One of the syntax changes impacted how you can create methods. The new version of JavaScript supports the old method syntax because many JavaScript developers still use it.

The new method syntax (see below) doesn't require arrow syntax or a colon (:) with the function keyword.

const restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'], openRestaurant() { return 'Unlock the door, flip the open sign. We are open for business!'; }, closeRestaurant() { return 'Lock the door, flip the open sign. We are closed.' }

In the example above we change the syntax for the .openRestaurant() and .closeRestaurant() methods. This syntax, which only requires the name of the method and parentheses, is best practice for defining methods.

# The this Keyword I

Objects hold data and functions, which we can use to represent real-world things in JavaScript.

The next step is to create methods that operate on the data inside of the same object.

Let's try it by using the hasDineInSpecialproperty in the .openRestaurant() method:

const restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'], openRestaurant() { if (hasDineInSpecial) { return 'Unlock the door, post the special on the board, then flip the open sign.'; } else { return 'Unlock the door, then flip the open sign.'; } } }; console.log(restaurant.openRestaurant());

The output would be:

ReferenceError: hasDineInSpecial is not defined

The error above doesn't work because hasDineInSpecial is out of the .openRestaurant() method's scope.

To address this scope issue, we can use the this keyword to access properties inside of the same object.

We can utilize this in the .openRestaurant() function as such:

const restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'], openRestaurant: function() { if (this.hasDineInSpecial) { return 'Unlock the door, post the special on the board, then flip the open sign.' } else { return 'Unlock the door, then flip the open sign.' } } } console.log(restaurant.openRestaurant());

The output would be:

Unlock the door, post the special on the board, then flip the open sign.

The .openRestaurant() method in the example above will return a value. The thiskeyword refers to the current object, which we use to grab the value saved to hasDineInSpecial.

this.hasDineInSpecial inside the object is the same as accessing restaurant.hasDineInSpecial outside the object.

# The this Keyword II

In Javascript, this refers to the object we call it inside.

For instance, if we have:

let myObj = { name: 'Miti', sayHello() { return `${this.name} says hello!`; } };

If we call myObj.sayHello(), our method will return 'Miti says hello!'. this in the example above is called inside the myObjobject, which limits the scope to the properties inside of myObj.

Let's change that by switching the object calling this:

let yourObj = { name: 'Timer' }; yourObj.sayHello = myObj.sayHello; // Sets the sayHello method on yourObj to be the sayHello method on yourObj

If you call yourObj.sayHello(), it will return 'Timer says hello!'. this in the example above is called inside the yourObj object, which limits the scope to the properties inside of yourObj.

Getters and Setters I

A common object design paradigm is to include *getter* and *setter* methods as attributes.

Getter and setter methods get and set the properties inside of an object. There are a couple of advantages to using these methods for getting and setting properties directly:

* You can check if new data is valid before setting a property.
* You can perform an action on the data while you are getting or setting a property.
* You can control which properties can be set and retrieved.

Let's consider our restaurant object from earlier:

let restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine Pesto'] }

In the example above, the seatingCapacitykey holds the number 120. Let's imagine that the restaurant managers like that this value is a number because they can use it to calculate the number of available seats at any given time during the evening.

Available seats = Capacity - Seats Taken

Imagine the restaurant adds an extra room to increase the seating capacity by thirty people and the managers must change the seatingCapacity value in the restaurant object.

A good developer would anticipate seatingCapacity as something that could change. To address this change, they would write code that checks if the newly set seatingCapacity value is valid. For example, the method should check if the seatingCapacity field is a number like 150, not the string 'one hundred fifty'. We can write this into a setter method as follows:

let restaurant = { \_name: 'Italian Bistro', \_seatingCapacity: 120, \_hasDineInSpecial: true, \_entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'], set seatingCapacity(newCapacity) { if (typeof newCapacity === 'number') { this.\_seatingCapacity = newCapacity; console.log(`${newCapacity} is valid input.`); } else { console.log(`Change ${newCapacity} to a number.`) } } }

Let's consider the new information in this example step-by-step.

* We prepended the property names with underscores (\_). Developers use an underscore before a property name to indicate a property or value should not be modified directly by other code. We recommend prepending all properties with an underscore, and creating setters for all attributes you want to access later in your code.
* The set seatingCapacity() setter method accepts newCapacity as a variable. The newCapacity variable holds the new value that we will store in \_seatingCapacity.
* Inside of the .seatingCapacity() setter we use a conditional statement to check if the newCapacity variable (our new value) is a number.
* If the input value is a number (valid input), then we use this.\_seatingCapacity to change the value assigned to \_seatingCapacity. If it is not valid, then we output a message to the user.

# Getters and Setters II

Now that you know how to create a setter method, you may be wondering how we use it. We call setter methods the same way we edited properties. The example below contains the same object as the last exercise.

let restaurant = { \_name: 'Italian Bistro', \_seatingCapacity: 120, \_hasDineInSpecial: true, \_entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'], set seatingCapacity(newCapacity) { if (typeof newCapacity === 'number') { this.\_seatingCapacity = newCapacity; console.log(`${newCapacity} is valid input.`); } else { console.log(`Change ${newCapacity} to a number.`); } }

The code below calls the setter method:

// Sets the \_seatingCapacity value to 150 restaurant.seatingCapacity = 150;

The output would be:

150 is valid input.

In the example above, we set the \_seatingCapacity value to 150. We use the same syntax we would use to set a property that doesn't have a setter method. Since the input (150) is a number, our method will execute the first block in the conditional statement — it changes \_seatingCapacity to 150 and logs 150 is a valid input. to the console.

# Getters and Setters III

Once you've set the properties, you need a way to access them. Getters are used to get the property values inside of an object.

let restaurant = { \_name: 'Italian Bistro', \_seatingCapacity: 120, \_hasDineInSpecial: true, \_entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'], set seatingCapacity(newCapacity) { if (typeof newCapacity === 'number') { this.\_seatingCapacity = newCapacity; } else { console.log(`Change ${newCapacity} to a number.`) } }, get seatingCapacity() { console.log(`There are ${this.\_seatingCapacity} seats at Italian Bistro.`); return this.\_seatingCapacity; } }

In the example above, the getter method (get seatingCapacity()) logs something to the console and returns the value saved to \_seatingCapacity. We call the getter method the same way we would access a property without a method:

restaurant.seatingCapacity = 150; const seats = restaurant.seatingCapacity;

In this example we set the seatingCapacity to 150, then call the getter method using restaurant.seatingCapacity and save the result to a variable called seats. The getter will also log the following line of code to the console:

There are 150 seats at Italian Bistro.

Review: Objects

Way to go! Let's review what we learned in this lesson:

* Objects store key-value pairs and let us represent real-world things in JavaScript.
* Properties in objects are separated by commas. Key-value pairs are always separated by a colon.
* You can add or edit a property within an object with dot notation.
* A method is a function in an object.
* this helps us with scope inside of object methods. this is a dynamic variable that can change depending on the object that is calling the method.
* Getter and setter methods allow you to process data before accessing or setting property values.