**Advanced Objects Introduction**

Remember, objects in JavaScript are containers that store data and functionality. In this lesson, we will build upon the fundamentals of creating objects and explore some advanced concepts.

So if there are no objections, let's learn more about objects!

In this lesson we will cover these topics:

* how to use the this keyword.
* conveying privacy in JavaScript methods.
* defining getters and setters in objects.
* creating factory functions.
* using destructuring techniques.

# The this Keyword

Objects are collections of related data and functionality. We store that functionality in methods on our objects:

const goat = { dietType: 'herbivore', makeSound() { console.log('baaa'); } };

In our goat object we have a .makeSound()method. We can invoke the .makeSound()method on goat.

goat.makeSound(); // Prints baaa

Nice, we have a goat object that can print baaato the console. Everything seems to be working fine. What if we wanted to add a new method to our goat object called .diet() that prints the goat's dietType?

const goat = { dietType: 'herbivore', makeSound(){ console.log('baaa'); }, diet() { console.log(dietType) } }; goat.diet(); // Output will be "ReferenceError: dietType is not defined"

That's strange, why is dietType not defined even though it's a property of goat? That's because inside the scope of the .diet()method, we don't automatically have access to other properties of the goat object.

Here's where the this keyword comes to the rescue. If we change the .diet() method to use the this, the .diet() works! :

const goat = { dietType: 'herbivore', makeSound(){ console.log('baaa'); }, diet() { console.log(this.dietType) } }; goat.diet(); // Output: herbivore

The this keyword references the calling objectwhich provides access to the calling object's properties. In the example above, the calling object is goat and by using this we're accessing the goat object itself, and then the dietType property of goat by using property dot notation.

Let's get comfortable using the this keyword in a method.

# Arrow Functions and this

We saw in the previous exercise that for a method, the calling object is the object the method belongs to. If we use the this keyword in a method then the value of this is the calling object. However, it becomes a bit more complicated when we start using arrow functions for methods. Take a look at the example below:

const goat = { dietType: 'herbivore', makeSound(){ console.log('baaa'); }, diet: () => { console.log(this.dietType) } }; goat.diet(); // Prints undefined

In the comment, you can see that goat.diet()would log undefined. So what happened? Notice that in the .diet() is defined using an arrow function.

Arrow functions inherently bind, or tie, an already defined this value to the function itself that is NOT the calling object. In the code snippet above, the value of this is the global object, or an object that exists in the global scope, which doesn't have a dietType property and therefore returns undefined.

To read more about either arrow functions or the global object check out the MDN documentation of [the global object](https://developer.mozilla.org/en-US/docs/Glossary/Global_object) and [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions).

The key takeaway from the example above is to avoid using arrow functions when using this in a method!

# Privacy

Accessing and updating properties is fundamental in working with objects. However, there are cases in which we don't want other code simply accessing and updating an object's properties. When discussing privacy in objects, we define it as the idea that only certain properties should be mutable or able change in value.

Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property. One common convention is to place an underscore \_ before the name of a property to mean that the property should not be altered. Here's an example of using \_ to prepend a property.

const bankAccount = { \_amount: 1000 }

In the example above, the \_amount is not intended to be directly manipulated.

Even so, it is still possible to reassign \_amount:

bankAccount.\_amount = 1000000;

In later exercises, we'll cover the use of methods called getters and setters. Both methods are used to respect the intention of properties prepended, or began, with \_. Getters can return the value of internal properties and setters can safely reassign property values. For now, let's see what happens if we can change properties that don't have setters or getters.

**Getters**

*Getters* are methods that get and return the internal properties of an object. But they can do more than just retrieve the value of a property! Let's take a look at a getter method:

const person = { \_firstName: 'John', \_lastName: 'Doe', get fullName() { if (this.\_firstName && this.\_lastName){ return `${this.\_firstName} ${this.\_lastName}`; } else { return 'Missing a first name or a last name.'; } } } // To call the getter method: person.fullName; // 'John Doe'

Notice that in the getter method above:

* We use the get keyword followed by a function.
* We use an if...else conditional to check if both \_firstName and \_lastName exist (by making sure they both return truthy values) and then return a different value depending on the result.
* We can access the calling object's internal properties using this. In fullName, we're accessing both this.\_firstName and this.\_lastName.
* In the last line we call fullName on person. In general, getter methods do not need to be called with a set of parentheses. Syntactically, it looks like we're accessing a property.

Now that we've gone over syntax, let's discuss what some notable advantages of using a getter method:

* Getters can perform an action on the data when getting a property.
* Getters can return different values using conditionals.
* In a getter, we can access the properties of the calling object using this.
* The functionality of our code is easier for other developers to understand.

Another thing to keep in mind when using getter (and setter) methods is that properties cannot share the same name as the getter/setter function. If we do so, then calling the method will result in an infinite call stack error. One workaround is to add an underscore before the property name like we did in the example above.

Great, let's go getter!

# Setters

Along with getter methods, we can also create setter methods which reassign values of existing properties within an object. Let's see an example of a setter method:

const person = { \_age: 37, set age(newAge){ if (typeof newAge === 'number'){ this.\_age = newAge; } else { console.log('You must assign a number to age'); } } };

Notice that in the example above:

* We can perform a check for what value is being assigned to this.\_age.
* When we use the setter method, only values that are numbers will reassign this.\_age
* There are different outputs depending on what values are used to reassign this.\_age.

Then to use the setter method:

person.age = 40; console.log(person.\_age); // Logs: 40 person.age = '40'; // Logs: You must assign a number to age

Setter methods like age do not need to be called with a set of parentheses. Syntactically, it looks like we're reassigning the value of a property.

Like getter methods, there are similar advantages to using setter methods that include checking input, performing actions on properties, and displaying a clear intention for how the object is supposed to be used. Nonetheless, even with a setter method, it is still possible to directly reassign properties. For example, in the example above, we can still set .\_age directly:

person.\_age = 'forty-five' console.log(person.\_age); // Prints forty-five

# Factory Functions

So far we've been creating objects individually, but there are times where we want to create many instances of an object quickly. Here's where factory functions come in. A real world factory manufactures multiple copies of an item quickly and on a massive scale. A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have parameters allowing us to customize the object that gets returned.

Let's say we wanted to create an object to represent monsters in JavaScript. There are many different types of monsters and we could go about making each monster individually but we can also use a factory function to make our lives easier. To achieve this diabolical plan of creating multiple monsters objects, we can use a factory function that has parameters:

const monsterFactory = (name, age, energySource, catchPhrase) => { return { name: name, age: age, energySource: energySource, scare() { console.log(catchPhrase); } } };

In the monsterFactory function above, it has four parameters and returns an object that has the properties: name, age, energySource, and scare(). To make an object that represents a specific monster like a ghost, we can call monsterFactory with the necessary arguments and assign the return value to a variable:

const ghost = monsterFactory('Ghouly', 251, 'ectoplasm', 'BOO!'); ghost.scare(); // 'BOO!'

Now we have a ghost object as a result of calling monsterFactory() with the needed arguments. With monsterFactory in place, we don't have to create an object literal every time we need a new monster. Instead, we can invoke the monsterFactory function with the necessary arguments to  make a monster for us!

# Property Value Shorthand

ES6 introduced some new shortcuts for assigning properties to variables known asdestructuring.

In the previous exercise, we created a factory function that helped us create objects. We had to assign each property a key and value even though the key name was the same as the parameter name we assigned to it. To remind ourselves, here's a truncated version of the factory function:

const monsterFactory = (name, age) => { return { name: name, age: age } };

Imagine if we had to include more properties, that process would quickly become tedious! But we can use a destructuring technique, called property value shorthand, to save ourselves some keystrokes. The example below works exactly like the example above:

const monsterFactory = (name, age) => { return { name, age } };

Notice that we don't have to repeat ourselves for property assignments!

# Destructured Assignment

We often want to extract key-value pairs from objects and save them as properties. Take for example the following object:

const vampire = { name: 'Dracula', residence: 'Transylvania', preferences: { day: 'stay inside', night: 'satisfy appetite' } };

If we wanted to extract the residence property as a variable, we could using the following code:

const residence = vampire.residence; console.log(residence); // Prints 'Transylvania'

However, we can also take advantage of a destructuring technique called destructured assignment to save ourselves some keystrokes. In destructured assignment we create a variable with the name of an object's key that is wrapped in curly braces { } and assign to it the object. Take a look at the example below:

const { residence } = vampire; console.log(residence); // Prints 'Transylvania'

Look back at the vampire object's properties in the first code example. Then, in the example above, we declare a new variable residencethat extracts the value of the residenceproperty of vampire. When we log the value of residence to the console, 'Transylvania' is printed.

We can even use destructured assignment to grab nested properties of an object:

const { day } = vampire.preferences; console.log(day); // Prints 'stay inside'

# Built-in Object Methods

In the previous exercises we've been creating instances of objects that have their own methods. But, we can also take advantage of built-in methods for Objects!

For example, we have access to object instance methods like: .hasOwnProperty(), .valueOf(), and many more! Practice your documentation reading skills and check out: [MDN's object instance documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#Methods).

There are also useful Object class methods such as Object.assign(), Object.entries(), and Object.keys() just to name a few. For a comprehensive list, browse: [MDN's object instance documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#Methods_of_the_Object_constructor).

Let's get acquainted with some of these methods and their documentation.

Note: You will see errors as you work through this exercise, but by the end the errors will be fixed!

# Review

Congratulations on finishing Advanced Objects!

Let's review the concepts covered in this lesson:

* The object that a method belongs to is called the calling object.
* The this keyword refers the calling object and can be used to access properties of the calling object.
* Methods do not automatically have access to other internal properties of the calling object.
* The value of this depends on where the this is being accessed from.
* We cannot use arrow functions as methods if we want to access other internal properties.
* JavaScript objects do not have built-in privacy, rather there are conventions to follow to notify other developers about the intent of the code.
* The usage of an underscore before a property name means that the original developer did not intend for that property to be directly changed.
* Setters and getter methods allow for more detailed ways of accessing and assigning properties.
* Factory functions allow us to create object instances quickly and repeatedly.
* There are different ways to use object destructuring: one way is the property value shorthand and another is destructured assignment.
* As with any concept, it is a good skill to learn how to use the documentation with objects!

You're ready to start leveraging more elegant code for creating and accessing objects in your code!