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.

**Instructions**

Look over the code in **main.js** to see examples of the object related concept covered in the lesson. Then click next to get started on learning these concepts!

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 baaa to 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 object which 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.

### Instructions

**1.**

Let’s create a new object to practice using this.

In **main.js** there is an object robot, add a property of model and assign to it a value of '1E78V2'. Add another property, energyLevel and assign to it a value of 100.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Inside the robot object, add a method named provideInfo. Inside the body of provideInfo(), return the following string by using interpolation:

I am MODEL and my current energy level is ENERGYLEVEL.

Replace ‘MODEL’ and ‘ENERGYLEVEL’ with the calling object’s model and energyLevel property. Remember, to get the access to the calling object’s properties inside a method, you have to use the this keyword!

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Now to check .provideInfo() has access to the internal properties of robot. Log the result of calling .provideInfo() method on robot to the console.

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!

### Instructions

**1.**

Currently the .checkEnergy() method is not producing the correct output because it is using arrow function syntax.

Refactor, or change, the method to use a function expression. You can write the function using either longhand or shorthand format.

After refactoring the method, notice that .checkEnergy() has access to the other internal properties of the robot object.

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 to 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.

### Instructions

**1.**

Below the robot object, reassign the \_energyLevel property to 'high'.

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 some notable advantages of using getter methods:

* 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!

**Instructions**

**1.**

In robot, create a getter method named energyLevel using the get keyword. Leave function body blank for now.

Stuck? Get a hint

**2.**

Inside the getter method, add an if statement to check if this.\_energyLevel is a number using the typeof operator. If that condition is truthy, return 'My current energy level is ENERGYLEVEL'. Replace ENERGYLEVEL with the value of this.\_energyLevel.

Make sure you return the string and not logging it to the console.

Stuck? Get a hint

**3.**

If this.\_energyLevel isn’t a number it could be that the \_energyLevel property was altered. Let’s add a default return statement for when such a scenario arises.

Add an else statement that returns 'System malfunction: cannot retrieve energy level'.

Stuck? Get a hint

**4.**

Log the result of calling the getter method energyLevel on robot to the console.

Notice that the method will return a formatted response rather than just accessing a property!

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

**Instructions**

**1.**

Currently, in robot there is a getter method for numOfSensors but no setter method! What if we need to add or remove some sensors? Let’s fix that problem.

Add a setter method named numOfSensors using the set keyword. Provide a parameter of num. Leave the function body empty for now.

Stuck? Get a hint

**2.**

There are a couple of things we should do in the setter method:

* Add a check to see if num is a number using the typeof operator.
* num should also be greater than or equal to 0.
* If both conditions are met, reassign this.\_numOfSensors to num.

Stuck? Get a hint

**3.**

Now add an else that logs 'Pass in a number that is greater than or equal to 0' to the console.

Stuck? Get a hint

**4.**

Use the numOfSensors setter method on robot to assign \_numOfSensors to 100.

Stuck? Get a hint

**5.**

To check that the setter method worked, console.log() robot.numOfSensors.

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!

### Instructions

**1.**

Instead of making individual robots like we’ve been doing, let’s make a factory function to make robots as we please!

Create a factory function named robotFactory that has two parameters model and mobile. Make the function return an object. In the object, add a key of model with the value of the model parameter. Add another property that has a key of mobile with a value of the mobile parameter.

Then add a method named beep without a parameter that will log 'Beep Boop' to the console.

Stuck? Get a hint

**2.**

Use your factory function by declaring a const variable named tinCan. Assign to tinCan the value of calling robotFactory with the first argument of 'P-500' and the second argument of true.

**3.**

Let’s now check what tinCan can do! Call .beep() on tinCan.

You should see 'Beep Boop' printed to the console which means the factory function produced a robot object! Play around with tinCan to check the other properties!