**Introduction to Objects**

It’s time to learn more about the basic structure that permeates nearly every aspect of JavaScript programming: objects.

You’re probably already more comfortable with objects than you think, because JavaScript loves objects! Many components of the language are actually objects under the hood, and even the parts that aren’t— like strings or numbers— can still act like objects in some instances.

There are only seven fundamental data types in JavaScript, and six of those are the primitive data types: string, number, boolean, null, undefined, and symbol. With the seventh type, objects, we open our code to more complex possibilities. We can use JavaScript objects to model real-world things, like a basketball, or we can use objects to build the data structures that make the web possible.

At their core, JavaScript objects are containers storing related data and functionality, but that deceptively simple task is extremely powerful in practice. You’ve been using the power of objects all along, but now it’s time to understand the mechanics of objects and start making your own!

**Creating Object Literals**

Objects can be assigned to variables just like any JavaScript type. We use curly braces, {}, to designate an *object literal*:

let spaceship = {}; // spaceship is an empty object

We fill an object with unordered data. This data is organized into *key-value pairs*. A key is like a variable name that points to a location in memory that holds a value.

A key’s value can be of any data type in the language including functions or other objects.

We make a key-value pair by writing the key’s name, or *identifier*, followed by a colon and then the value. We separate each key-value pair in an object literal with a comma (,). Keys are strings, but when we have a key that does not have any special characters in it, JavaScript allows us to omit the quotation marks:

// An object literal with two key-value pairs

let spaceship = {

'Fuel Type': 'diesel',

color: 'silver'

};

The spaceship object has two properties Fuel Type and color. 'Fuel Type' has quotation marks because it contains a space character.

Let’s make some objects!

**Instructions**

**1.**

The spaceship we have so far looks good but, unfortunately, is not very fast at hyperspace travel due to having an inferior fuel source. Make a new spaceship object named fasterShip with the same color as spaceship but with a Fuel Type equal to 'Turbo Fuel'.

Hint

Create an object literal with two properties.

let objectName = {

'Property Name': 'Property Value',

propName: 'Prop Value'

};

let fasterShip = {

  'Fuel Type': 'Turbo Fuel',

  color: 'silver'

};

# Accessing Properties

There are two ways we can access an object’s property. Let’s explore the first way— dot notation, ..

You’ve used dot notation to access the properties and methods of built-in objects and data instances:

'hello'.length; // Returns 5

With property dot notation, we write the object’s name, followed by the dot operator and then the property name (key):

let spaceship = {

homePlanet: 'Earth',

color: 'silver'

};

spaceship.homePlanet; // Returns 'Earth',

spaceship.color; // Returns 'silver',

If we try to access a property that does not exist on that object, undefined will be returned.

spaceship.favoriteIcecream; // Returns undefined

Let’s get some more practice using dot notation on an object!

**Instructions**

**1.**

Let’s use the dot operator to access the value of numCrew from the spaceship object in the code editor. Create a variable crewCount and assign the spaceship‘s numCrew property to it.

Hint

Create a variable with a value of the object’s property:

const variableName = objectName.propertyName;

**2.**

Again using the dot operator, create a variable planetArray and assign the spaceship‘s flightPath property to it.

Hint

Create a variable with a value of the object’s property:

const variableName = objectName.propertyName;

let spaceship = {

  homePlanet: 'Earth',

  color: 'silver',

  'Fuel Type': 'Turbo Fuel',

  numCrew: 5,

  flightPath: ['Venus', 'Mars', 'Saturn']

};

// Write your code below

let crewCount = spaceship.numCrew;

let planetArray = spaceship.flightPath;

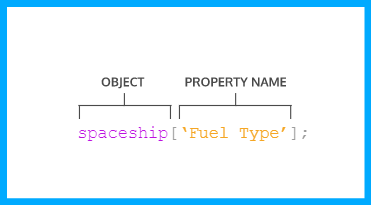
# Bracket Notation

The second way to access a key’s value is by using bracket notation, [ ].

You’ve used bracket notation when indexing an array:

['A', 'B', 'C'][0]; // Returns 'A'

To use bracket notation to access an object’s property, we pass in the property name (key) as a string.



We \*must\* use bracket notation when accessing keys that have numbers, spaces, or special characters in them. Without bracket notation in these situations, our code would throw an error.

let spaceship = {

'Fuel Type': 'Turbo Fuel',

'Active Duty': true,

homePlanet: 'Earth',

numCrew: 5

};

spaceship['Active Duty']; // Returns true

spaceship['Fuel Type']; // Returns 'Turbo Fuel'

spaceship['numCrew']; // Returns 5

spaceship['!!!!!!!!!!!!!!!']; // Returns undefined

With bracket notation you can also use a variable inside the brackets to select the keys of an object. This can be especially helpful when working with functions:

let returnAnyProp = (objectName, propName) => objectName[propName];

returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'

If we tried to write our returnAnyProp() function with dot notation (objectName.propName) the computer would look for a key of 'propName' on our object and not the value of the propName parameter.

Let’s get some practice using bracket notation to access properties!

**Instructions**

**1.**

Let’s use bracket notation to access the value of 'Active Mission' from the spaceship object in the code editor. Create a variable isActive and assign the spaceship‘s 'Active Mission' property to it.

Hint

let variableName = objectName['propertyName']

**2.**

Using bracket notation and the propName variable provided, console.log() the value of the 'Active Mission' property.

Hint

console.log(objectName[variableName])

let spaceship = {

  'Fuel Type' : 'Turbo Fuel',

  'Active Mission' : true,

  homePlanet : 'Earth',

  numCrew: 5

 };

let propName =  'Active Mission';

// Write your code below

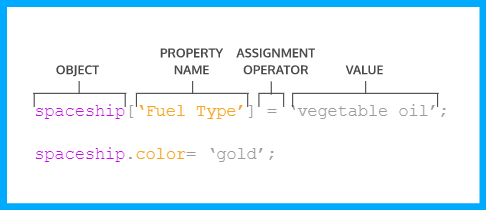
let isActive = spaceship['Active Mission'];

console.log(spaceship[propName]);

**Property Assignment**

Once we’ve defined an object, we’re not stuck with all the properties we wrote. Objects are *mutable* meaning we can update them after we create them!

We can use either dot notation, ., or bracket notation, [], and the assignment operator, = to add new key-value pairs to an object or change an existing property.



One of two things can happen with property assignment:

* If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.
* If there was no property with that name, a new property will be added to the object.

It’s important to know that although we can’t reassign an object declared with const, we can still mutate it, meaning we can add new properties and change the properties that are there.

const spaceship = {type: 'shuttle'};

spaceship = {type: 'alien'}; // TypeError: Assignment to constant variable.

spaceship.type = 'alien'; // Changes the value of the type property

spaceship.speed = 'Mach 5'; // Creates a new key of 'speed' with a value of 'Mach 5'

You can delete a property from an object with the delete operator.

const spaceship = {

'Fuel Type': 'Turbo Fuel',

homePlanet: 'Earth',

mission: 'Explore the universe'

};

delete spaceship.mission; // Removes the mission property

**Instructions**

**1.**

Reassign the color property of the spaceship object to have a value of 'glorious gold'

Hint

objectName['Property Name'] = 'New Property Value';

objectName.propName = 'New Prop Value';

**2.**

Without changing lines 1 - 6, add a numEngines property with a numeric value between 1 and 10 to the spaceship object.

Hint

objectName['Property Name'] = 'New Property Value';

objectName.propName = 'New Prop Value';

**3.**

Use the delete operator to remove the 'Secret Mission' property from the spaceship object.

Hint

delete objectName['Property Name'];

delete objectName.propName;

let spaceship = {

  'Fuel Type' : 'Turbo Fuel',

  homePlanet : 'Earth',

  color: 'silver',

  'Secret Mission' : 'Discover life outside of Earth.'

};

// Write your code below

 spaceship.color = 'glorious gold';

spaceship['numEngines'] = 6;

delete spaceship['Secret Mission'];

# Methods

When the data stored on an object is a function we call that a method. A property is what an object has, while a method is what an object does.

Do object methods seem familiar? That’s because you’ve been using them all along! For example console is a global javascript object and .log() is a method on that object. Math is also a global javascript object and .floor() is a method on it.

We can include methods in our object literals by creating ordinary, comma-separated key-value pairs. The key serves as our method’s name, while the value is an anonymous function expression.

const alienShip = {

invade: function () {

console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')

}

};

With the new method syntax introduced in ES6 we can omit the colon and the function keyword.

const alienShip = {

invade () {

console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')

}

};

Object methods are invoked by appending the object’s name with the dot operator followed by the method name and parentheses:

alienShip.invade(); // Prints 'Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.'

**Instructions**

**1.**

Below the retreatMessage variable in the code editor, create an alienShip object. It should contain a method .retreat() which will console.log() the retreatMessage.

Hint

let objectName = {

methodName() {

console.log('The methodName method was invoked!')

}

};

**2.**

Add another method to your object literal. This method, .takeOff(), should console.log() the string 'Spim... Borp... Glix... Blastoff!'.

Hint

Don’t forget to separate your methods with commas just as you would any other key-value pairs:

let objectName = {

methodName() {

console.log('The methodName method was invoked!')

},

secondMethodName() {

console.log('The secondMethodName method was invoked!')

}

};

**3.**

Invoke your two methods: first .retreat() then .takeOff().

Hint

objectName.methodName()

let retreatMessage = 'We no longer wish to conquer your planet. It is full of dogs, which we do not care for.';

// Write your code below

let alienShip = {

  retreat() {

    console.log(retreatMessage)

  },

  takeOff() {

    console.log('Spim... Borp... Glix... Blastoff!')

  }

};

alienShip.retreat();

alienShip.takeOff();

**Nested Objects**

In application code, objects are often nested— an object might have another object as a property which in turn could have a property that’s an array of even more objects!

In our spaceship object, we want a crew object. This will contain all the crew members who do important work on the craft. Each of those crew members are objects themselves. They have properties like name, and degree, and they each have unique methods based on their roles. We can also nest other objects in the spaceship such as a telescope or nest details about the spaceship’s computers inside a parent nanoelectronics object.

const spaceship = {

telescope: {

yearBuilt: 2018,

model: '91031-XLT',

focalLength: 2032

},

crew: {

captain: {

name: 'Sandra',

degree: 'Computer Engineering',

encourageTeam() { console.log('We got this!') }

}

},

engine: {

model: 'Nimbus2000'

},

nanoelectronics: {

computer: {

terabytes: 100,

monitors: 'HD'

},

'back-up': {

battery: 'Lithium',

terabytes: 50

}

}

};

We can chain operators to access nested properties. We’ll have to pay attention to which operator makes sense to use in each layer. It can be helpful to pretend you are the computer and evaluate each expression from left to right so that each operation starts to feel a little more manageable.

spaceship.nanoelectronics['back-up'].battery; // Returns 'Lithium'

In the preceding code:

* First the computer evaluates spaceship.nanoelectronics, which results in an object containing the back-up and computer objects.
* We accessed the back-up object by appending ['back-up'].
* The back-up object has a battery property, accessed with .battery which returned the value stored there: 'Lithium'

**Instructions**

**1.**

Create a variable capFave and assign the captain‘s favorite food (the element in the 0th index of her 'favorite foods' array) to it. Make sure to use bracket and dot notation to get the value of the food through nested access (don’t just copy the value into the variable!)

Hint

spaceship.crew.captain['favorite foods'] will give us access to the array of the captain‘s favorite foods, but there’s one additional step to get the first item in that array.

**2.**

Right now the passengers property has a value of null. Instead, assign as its value an array of objects. These objects should represent the spaceship‘s passengers as individual objects. Make at least one passenger object in the array that has at least one key-value pair on it.

Hint

passengers : [{name: 'Space Dog'}]

**3.**

Create a variable firstPassenger and assign the first passenger as its value (the element in the 0th index of the spaceship.passengers array you just made). Make sure to use bracket and dot notation to get the passenger object through nested access (don’t just copy the object into the variable!)

Hint

spaceship.passengers[0]

let spaceship = {

  passengers: [{name: 'Space Dog'}],

  telescope: {

    yearBuilt: 2018,

    model: "91031-XLT",

    focalLength: 2032

  },

  crew: {

    captain: {

      name: 'Sandra',

      degree: 'Computer Engineering',

      encourageTeam() { console.log('We got this!') },

     'favorite foods': ['cookies', 'cakes', 'candy', 'spinach'] }

  },

  engine: {

    model: "Nimbus2000"

  },

  nanoelectronics: {

    computer: {

      terabytes: 100,

      monitors: "HD"

    },

    'back-up': {

      battery: "Lithium",

      terabytes: 50

    }

  }

};

let capFave = spaceship.crew.captain['favorite foods'][0];

let firstPassenger = spaceship.passengers[0];

**Pass By Reference**

Objects are *passed by reference*. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a const variable).

const spaceship = {

homePlanet : 'Earth',

color : 'silver'

};

let paintIt = obj => {

obj.color = 'glorious gold'

};

paintIt(spaceship);

spaceship.color // Returns 'glorious gold'

Our function paintIt() permanently changed the color of our spaceship object. However, reassignment of the spaceship variable wouldn’t work in the same way:

let spaceship = {

homePlanet : 'Earth',

color : 'red'

};

let tryReassignment = obj => {

obj = {

identified : false,

'transport type' : 'flying'

}

console.log(obj) // Prints {'identified': false, 'transport type': 'flying'}

};

tryReassignment(spaceship) // The attempt at reassignment does not work.

spaceship // Still returns {homePlanet : 'Earth', color : 'red'};

spaceship = {

identified : false,

'transport type': 'flying'

}; // Regular reassignment still works.

Let’s look at what happened in the code example:

* We declared this spaceship object with let. This allowed us to reassign it to a new object with identified and 'transport type' properties with no problems.
* When we tried the same thing using a function designed to reassign the object passed into it, the reassignment didn’t stick (even though calling console.log() on the object produced the expected result).
* When we passed spaceship into that function, obj became a reference to the memory location of the spaceship object, but *not* to the spaceship variable. This is because the obj parameter of the tryReassignment() function is a variable in its own right. The body of tryReassignment() has no knowledge of the spaceship variable at all!
* When we did the reassignment in the body of tryReassignment(), the obj variable came to refer to the memory location of the object {'identified' : false, 'transport type' : 'flying'}, while the spaceship variable was completely unchanged from its earlier value.

**Instructions**

**1.**

Write a function greenEnergy() that has an object as a parameter and sets that object’s 'Fuel Type' property to 'avocado oil'.

Hint

let functionName = objectParam => {

objectParam['Property Name'] = 'New Property Value';

};

**2.**

Write a function remotelyDisable() that has an object as a parameter and sets (or reassigns) that object’s disabled property to true.

Hint

let functionName = objectParam => {

objectParam.propertyName = 'A Property Value';

};

**3.**

Call your two functions with the spaceship object in the code editor, then console.log() the spaceship object to confirm those properties were changed/added.

let spaceship = {

  'Fuel Type' : 'Turbo Fuel',

  homePlanet : 'Earth'

};

// Write your code below

const greenEnergy = object =>{

  object['Fuel Type'] = 'avocado oil';

};

const remotelyDisable = obj =>{

  obj['disabled']=true;

};

remotelyDisable(spaceship);

greenEnergy(spaceship);

console.log(spaceship);

# Looping Through Objects

Loops are programming tools that repeat a block of code until a condition is met. We learned how to iterate through arrays using their numerical indexing, but the key-value pairs in objects aren’t ordered! [JavaScript has given us alternative solution for iterating through objects with the for...in syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in).

for...in will execute a given block of code for each property in an object.

let spaceship = {

crew: {

captain: {

name: 'Lily',

degree: 'Computer Engineering',

cheerTeam() { console.log('You got this!') }

},

'chief officer': {

name: 'Dan',

degree: 'Aerospace Engineering',

agree() { console.log('I agree, captain!') }

},

medic: {

name: 'Clementine',

degree: 'Physics',

announce() { console.log(`Jets on!`) } },

translator: {

name: 'Shauna',

degree: 'Conservation Science',

powerFuel() { console.log('The tank is full!') }

}

}

};

// for...in

for (let crewMember in spaceship.crew) {

console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)

};

Our for...in will iterate through each element of the spaceship.crew object. In each iteration, the variable crewMember is set to one of spaceship.crew‘s keys, enabling us to log a list of crew members’ role and name.

**Instructions**

**1.**

Using for...in, iterate through the spaceship.crew object in the code editor and console.log() a list of crew roles and names in the following format: '[crew member's role]: [crew member's name]', e.g.,'chief officer: Dan'.

Hint

for (let variableName in outerObject.innerObject) {

console.log(`${variableName}: ${outerObject.innerObject[variableName].propertyName}`)

};

**2.**

Using for...in, iterate through the spaceship.crew object in the code editor and console.log() a list of crew names and degrees in the following format: '[crew member's name]: [crew member's degree]', i.e.,'Lily: Computer Engineering'.

Hint

for (let variableName in outerObject.innerObject) {

console.log(`${outerObject.innerObject[variableName].propertyName}: ${outerObject.innerObject[variableName].differentPropertyName}`)

};

let spaceship = {

    crew: {

    captain: {

        name: 'Lily',

        degree: 'Computer Engineering',

        cheerTeam() { console.log('You got this!') }

        },

    'chief officer': {

        name: 'Dan',

        degree: 'Aerospace Engineering',

        agree() { console.log('I agree, captain!') }

        },

    medic: {

        name: 'Clementine',

        degree: 'Physics',

        announce() { console.log(`Jets on!`) } },

    translator: {

        name: 'Shauna',

        degree: 'Conservation Science',

        powerFuel() { console.log('The tank is full!') }

        }

    }

};

// Write your code below

for (let crewMember in spaceship.crew) {

  console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)

};

for (let crewMember in spaceship.crew) {

  console.log(`${spaceship.crew[crewMember].name}: ${spaceship.crew[crewMember].degree}`)

};

# Review

Way to go! You’re well on your way to understanding the mechanics of objects in JavaScript. By building your own objects, you will have a better understanding of how JavaScript built-in objects work as well. You can also start imagining organizing your code into objects and modeling real world things in code.

Let’s review what we learned in this lesson:

* Objects store collections of key-value pairs.
* Each key-value pair is a property—when a property is a function it is known as a method.
* An object literal is composed of comma-separated key-value pairs surrounded by curly braces.
* You can access, add or edit a property within an object by using dot notation or bracket notation.
* We can add methods to our object literals using key-value syntax with anonymous function expressions as values or by using the new ES6 method syntax.
* We can navigate complex, nested objects by chaining operators.
* Objects are mutable—we can change their properties even when they’re declared with const.
* Objects are passed by reference— when we make changes to an object passed into a function, those changes are permanent.
* We can iterate through objects using the For...in syntax.

**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.

Hint

To add a property, you’re adding another key-value pair to an object. Remember to separate the properties by a comma.

**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!

Hint

If you don’t use this, you will get a reference error. Look at the following example for how to use this:

const goat = {

name: 'Billy',

color: 'biege',

giveDetails(){

return `${this.name} is a ${this.color} goat.`

}

};

goat.giveDetails(); // 'Billy is a biege goat.'

**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.

Hint

To invoke a method on an object, use dot notation followed by the name of the method with a set of parentheses ().

const robot = {

  model: '1E78V2',

  energyLevel: 100,

  provideInfo() {

    return `I am ${this.model} and my current energy level is ${this.energyLevel}.` ;

  }

};

console.log(robot.provideInfo());

# 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.

Hint

You can use the ES6 shorthand:

const goat = {

name: 'Billy',

color: 'biege',

giveDetails(){

console.log(`${this.name} is a ${this.color} goat.`)

}

}

Or the longhand format:

const goat = {

name: 'Billy',

color: 'biege',

giveDetails: function() {

console.log(`${this.name} is a ${this.color} goat.`)

}

}

const robot = {

  energyLevel: 100,

  checkEnergy() {

    console.log(`Energy is currently at ${this.energyLevel}%.`)

  }

}

robot.checkEnergy();

# 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'.

Hint

To reassign a property, you can use dot notation to access the property and then use the = operator to assign it to another value.

**2.**

Now take a look at the new recharge method in robot. .recharge() will add 30 to \_energyLevel.

What will happen now that \_energyLevel isn’t a number?

Call .recharge() on robot to find out.

Notice that a funky string is printed to the console! This is known as a side-effect of type-coercion. No need to worry about what this means for now, but it’s important to understand that you can cause unwanted side-effects when mutating objects and their properties.

const robot = {

  \_energyLevel: 100,

  recharge(){

    this.\_energyLevel += 30;

    console.log(`Recharged! Energy is currently at ${this.\_energyLevel}%.`)

  }

};

robot.\_energyLevel = 'high';

robot.recharge();

**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.

Hint

The robot object should now look like:

const robot = {

\_model: '1E78V2',

\_energyLevel: 65,

get energyLevel() {

}

}

Remember to separate the properties by commas.

**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.

Hint

When we use the typeof operator on a variable will return a string that contains the name of the data type. If the string is 'number' then the variable is a number datatype.

let test = 42

typeof test // 'number'

**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'.

Hint

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

**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!

Hint

To call a getter method, it looks syntactically like accessing a property. You do not need to include a set of parentheses.

robot.energyLevel;

Remember to log the result to the console.

console.log(robot.energyLevel);

const robot = {

  \_model: '1E78V2',

  \_energyLevel: 100,

  get energyLevel(){

    if(typeof this.\_energyLevel === 'number') {

      return 'My current energy level is ' + this.\_energyLevel

    } else {

      return "System malfunction: cannot retrieve energy level"

    }

  }

};

console.log(robot.energyLevel);

**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.

Hint

Remember to separate your properties using commas. All setters need at least one parameter, to provide a parameter for a setter use the following syntax:

const robot = {

\_numOfSensors: 15,

set numOfSensors(num) {

// Empty for now

}

};

**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.

Hint

You can add another conditional in an if statement using the && operator. If we look at just the property and setter method of robot, it should look like:

const robot = {

\_numOfSensors: 15,

set numOfSensors(num) {

if (typeof num === 'number' && num >= 0){

this.\_numOfSensors = num;

}

}

};

**3.**

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

Hint

If we look at just the property and setter method of robot, it should look like:

const robot = {

\_numOfSensors: 15;

set numOfSensors(num) {

if (typeof num === 'number' && num >= 0){

this.\_numOfSensors = num;

} else {

console.log('Pass in a number that is greater than or equal to 0');

}

}

}

**4.**

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

Hint

Using the setter method looks syntactically like reassigning a property.

robot.someSetterMethod = 9001;

**5.**

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

Hint

Since we have a getter method in place, we can use the getter method to access the \_numOfSensors property of robot.

Syntactically, the getter method looks like we’re accessing a property:

console.log(robot.numOfSensors);

const robot = {

  \_model: '1E78V2',

  \_energyLevel: 100,

  \_numOfSensors: 15,

  get numOfSensors(){

    if(typeof this.\_numOfSensors === 'number'){

      return this.\_numOfSensors;

    } else {

      return 'Sensors are currently down.'

    }

  },

  set numOfSensors(num) {

    if (typeof num === 'number' && num >= 0){

      this.\_numOfSensors = num;

    } else {

      console.log('Pass in a number that is greater than or equal to 0')

    }

  }

};

robot.numOfSensors = 100;

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.

Hint

Remember to separate your properties using commas. Your factory function will look like:

const robotFactory = (model, mobile) => {

return {

model: model,

mobile: mobile,

}

}

**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!

Hint

The test for this checkpoint will check if you’ve called .beep() on tinCan. Checking the other methods on tinCan is not tested for but will help you understand that robotFactory() created an object for us. You can even make another object using robotFactory()!

const robotFactory = (model, mobile) => {

  return {

    model : model,

    mobile: mobile,

    beep () {

      console.log('Beep Boop');

    }

  };

};

const tinCan = robotFactory('P-500', true);

tinCan.beep();

# Property Value Shorthand

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

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!

**Instructions**

**1.**

Use the property value shorthand and refactor the factory function in **main.js**

Hint

Remember to separate your properties with commas.

In robotFactory there are a few instances where the property’s key and value share the same name, you can remove the key : portion to destructure the object.

function robotFactory(model, mobile){

  return {

    model,

    mobile,

    beep() {

      console.log('Beep Boop');

    }

  }

}

// To check that the property value shorthand technique worked:

const newRobot = robotFactory('P-501', false)

console.log(newRobot.model)

console.log(newRobot.mobile)

# Destructured Assignment

We often want to extract key-value pairs from objects and save them as variables. 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 residence that extracts the value of the residence property 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'

**Instructions**

**1.**

Use destructured assignment to create a const variable named functionality that extracts the functionality property of robot.

If you need a reminder on how to use destructured assignment, review the example in the narrative or check the hint.

Hint

The syntax of using destructured assignment will look like:

const { propertyName } = obj;

**2.**

Since functionality is referencing robot.functionality we can call the methods available to robot.functionality simply through functionality.

Take advantage of this shortcut and call the .beep() method on functionality.

Hint

You can think of functionality as the object that was pulled out of robot.functionality. To call .beep(), use dot notation with the name of the method and a set of parentheses:

functionality.beep();

const robot = {

  model: '1E78V2',

  energyLevel: 100,

  functionality: {

    beep() {

      console.log('Beep Boop');

    },

    fireLaser() {

      console.log('Pew Pew');

    },

  }

};

const { functionality } = robot;

functionality.beep();

# 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!

**Instructions**

**1.**

In **main.js** there is an object, robot. We’d like to grab the property names, otherwise known as keys, and save the keys in an array which is assigned to robotKeys. However, there’s something missing in the method call.

Find out what we have to include by reading [MDN’s Object.keys() documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys).

Hint

If you look at the example at MDN, it looks like Object.keys() takes an argument of an object instance.

**2.**

Object.entries() will also return an array, but the array will contain more arrays that have both the key and value of the properties in an object.

Declare a const variable named robotEntries and assign to it the entries of robot by calling Object.entries().

To find how to use Object.entries(), read [the documentation at MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries).

Hint

Object.entries() is similar to Object.keys() in how it is called and what arguments it will take. Remember to check with the documentation!

**3.**

Now what if we want another object that has the properties of robot but with a few additional properties. Object.assign() sounds like a great method to use, but like the previous examples we should check [Object.assign() documentation at MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign" \t "_blank).

Declare a const variable named newRobot. newRobot will be a new object that has all the properties of robot and the properties in the following object: {laserBlaster: true, voiceRecognition: true}. Make sure that you are **not** changing the robot object!

Hint

When you use Object.assign() it is important to know which argument is the target or source(s). Take a look at [MDN’s syntax section](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#Syntax) and play around with the examples at the top to fully understand how to use Object.assign().

const robot = {

  model: 'SAL-1000',

  mobile: true,

  sentient: false,

  armor: 'Steel-plated',

  energyLevel: 75

};

// What is missing in the following method call?

const robotKeys = Object.keys(robot);

console.log(robotKeys);

// Declare robotEntries below this line:

const robotEntries = Object.entries(robot)

console.log(robotEntries);

// Declare newRobot below this line:

const newRobot = Object.assign({laserBlaster: true, voiceRecognition: true}, robot);

console.log(newRobot);

**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!

**Instructions**

If you want to challenge yourself:

* Find the value of this in a function inside of a method.
* Learn the outcome of using a property that has the exact same name as a setter/getter method.
* Create a new factory function that can create object instances of your choice.
* Read documentation on other destructuring techniques and apply it to your code.
* Try out other built-in object methods and learn what they do.