



# Modifying Objects

 <https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-modifying-objects>  <https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-modifying-objects/issues/new>

## Learning Goals

- Add an **Object** property using dot or bracket notation
- Modify a property using dot or bracket notation
- Update an **Object** nondestructively using the spread operator
- Remove a property from an **Object**
- Identify the relationship between **Array** s and **Object** s

## Introduction

In the previous lesson, we learned the basics of creating **Object** s and accessing their properties. In this lesson we'll learn how to modify and remove properties, both destructively and nondestructively. Finally, we'll explore the relationship between **Array** s and **Object** s.

As always, don't forget to follow along in [replit](https://replit.com/languages/javascript)  <https://replit.com/languages/javascript>.

## Add an **Object** Property Using Dot or Bracket Notation

We know how to initialize a variable by declaring it and assigning it a value using the assignment operator:

```
const city = "New York";
```

The process of creating a property inside an existing object is similar; we specify the key and assign it a value:

```
const circle = {}; // Create `circle` and set it to an empty Object
```

```
circle;
```

```
//=> {}
```

```
circle.radius = 5; // Create the key inside `circle` and set its value to 5
```

```
circle;
```

```
//=> { radius: 5 }
```

We can do this using either dot or bracket notation, and we can use any expression as the value:

```
const circle = {};
```

```
circle.radius = 5;
```

```
circle["diameter"] = 10;
```

```
circle.circumference = circle.diameter * Math.PI;
```

```
//=> 31.41592653589793
```

```
circle["area"] = Math.PI * circle.radius ** 2;
```

```
//=> 78.53981633974483
```

```
circle;
```

```
//=> { radius: 5, diameter: 10, circumference: 31.41592653589793, area: 78.53981633974483 }
```

**A Side Note:** Recall from the lesson on **Array** s that we can add, modify or delete elements even if we use **const** to initialize the **Array** . The same thing applies here: we can add, modify or delete properties, but we can't reassign the variable itself.

**Top Tip:** Note that the process above gives us an alternative to typing out our **Object** using literal syntax: we can initialize an empty object and then use dot notation or bracket notation to create the properties programmatically. This approach is less error-prone than using literal syntax since JavaScript creates the correct **Object** syntax for us. Try it out in the REPL.

## Modify a Property Using Dot or Bracket Notation

We can update or overwrite existing properties simply by assigning a new value to an existing key:

```
const mondayMenu = {
  cheesePlate: {
    soft: "Chèvre",
    semiSoft: "Gruyère",
    hard: "Manchego",
  },
  fries: "Curly",
  salad: "Cobb",
};

mondayMenu.fries = "Sweet potato";

mondayMenu;
//=> { cheesePlate: { soft: "Chèvre", semiSoft: "Gruyère", hard: "Manchego" }, fries: "Sweet potato", salad: "Cobb" }
```

Note that modifying an **Object**'s properties in the way we did above is *destructive*. This means that we're making changes directly to the original **Object**.

Let's take a look at an example. We'll start by creating a function to encapsulate this modification process:

```
function destructivelyUpdateObject(obj, key, value) {
  obj[key] = value; //Why are we using bracket notation here?

  return obj;
}
```

Our function takes three arguments: the original menu **Object**, the **key** identifying the property we want to update, and the **value** we want to change its value to.

At our restaurant, we've finished serving for the day. It's time to update our **mondayMenu** to the **tuesdayMenu**:

```
const mondayMenu = {
  cheesePlate: {
    soft: "Chèvre",
    semiSoft: "Gruyère",
    hard: "Manchego",
  },
  fries: "Sweet potato",
  salad: "Cobb",
};

const tuesdayMenu = destructivelyUpdateObject(mondayMenu, "salad", "Caesar");

tuesdayMenu;
//=> { cheesePlate: { soft: "Chèvre", semiSoft: "Gruyère", hard: "Manchego" }, fries: "Sweet potato", salad: "Caesar" }

tuesdayMenu.salad;
//=> "Caesar"
```

Looks like our `tuesdayMenu` came out perfectly. But what about `mondayMenu` ?

```
mondayMenu.salad;
//=> "Caesar"
```

Dang! We don't serve Caesar salad on Mondays. Instead of destructively updating the original menu, is there a way to nondestructively merge the change(s) into a new `Object`, leaving the original intact?

## Update an Object Nondestructively Using the Spread Operator

Let's create a new method; it will take the same three arguments as the previous method:

```
function nondestructivelyUpdateObject(obj, key, value) {  
  // Code to return new, updated menu object  
}
```

Recall from the lessons on **Array** s that we can use the **spread operator** [↗\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_syntax#spread\\_in\\_object\\_literals\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_object_literals) to copy all the elements of an existing array into a new array. We can do the same thing with **Object** s. Let's use the spread operator to copy all of the old menu **Object** 's properties into a new **Object** :

```
function nondestructivelyUpdateObject(obj, key, value) {  
  const newObj = { ...obj };  
  
  // Code to return new, updated menu object goes here  
}
```

This will create a clone of the original object and save it into a new variable. We can then update the newly-created **newObj** with the desired change and return that updated menu, leaving the original menu **Object** unchanged:

```
function nondestructivelyUpdateObject(obj, key, value) {  
  const newObj = { ...obj };  
  
  newObj[key] = value;  
  
  return newObj;  
}  
  
const sundayMenu = nondestructivelyUpdateObject(  
  tuesdayMenu,  
  "fries",  
  "Shoestring"  
);
```

```
tuesdayMenu.fries;  
//=> "Sweet potato"  
  
sundayMenu.fries;  
//=> "Shoestring"
```

To review, we are calling our `nondestructivelyUpdateObject()` function, passing as our arguments the original menu ( `tuesdayMenu` ) and the key and value representing the desired change. The function first makes a copy of `tuesdayMenu` , then changes the value associated with the `fries` key to `"Shoestring"` . Finally, it returns the updated menu, which is stored into the variable `sundayMenu` .

We can refactor the function to be a bit more concise and still achieve the same outcome:

```
function nondestructivelyUpdateObject(obj, key, value) {  
  return {  
    ...obj,  
    [key]: value,  
  };  
}  
  
const sundayMenu = nondestructivelyUpdateObject(  
  tuesdayMenu,  
  "fries",  
  "Shoestring"  
);  
  
tuesdayMenu.fries;  
//=> "Sweet potato"  
  
sundayMenu.fries;  
//=> "Shoestring"
```

Here, we're still returning a new object that has all the key-value pairs from the original object copied into it. We're also using bracket notation ( `[]` ) to dynamically assign a key on the object being returned with the value passed as an argument.

**Note:** The spread operator has been around for a while, but there's still a chance you'll encounter similar code written using another method, `Object.assign()` [↗\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/assign\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign). Like the spread operator, `Object.assign` allows us to combine properties from multiple `Object`s into a single `Object`.

In other languages (like Ruby), this behavior is called "merging." You take an original base `Object` (maybe with some typical "standard" attribute / value pairs already set), and then you "merge" in additional `Object`(s).

**NOTE:** Doing nondestructive updates (i.e. "creating new things and merging on top") is a really important pattern. It turns out that, in many places, nondestructive updates are more performant. The main reason for this is when you add something to an existing `Object`, the computer has to make sure that the `Object` has enough room to add what you're saying to add. If it doesn't, the computer needs to do cleanup work, find some more space, copy the old thing over, add the new thing, and then resume work. That "accounting" process is actually quite slow.

## Remove a Property from an Object

Uh oh, we ran out of Southwestern dressing, so we have to take the salad off the menu. In JavaScript, that's as easy as:

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

delete wednesdayMenu.salad;
//=> true

wednesdayMenu;
//=> { cheesePlate: { soft: "Brie", semiSoft: "Fontina", hard: "Provolone" }, fries: "Sweet potato" }
```

We pass the property that we'd like to remove to the `delete` [🔗\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) operator, and JavaScript takes care of the rest. Poof! No more `salad` property on the `wednesdayMenu` `Object`.

## Identify the Relationship Between Arrays and Objects

Think back to the early lesson on data types in JavaScript. We listed off seven types into which all data falls: numbers, strings, booleans, symbols, `Object`s, `null`, and `undefined`. Notice anything missing? Arrays!

Why isn't an `Array` a fundamental data type in JavaScript? The answer is that **it's actually a special type of `Object`**. Yes, that's right: **`Array`s are `Object`s**. To underscore this point, check out what the `typeof` operator returns when we use it on an `Array`:

```
typeof [];  
//=> "object"
```

We can set properties on an `Array` just like a regular `Object`:

```
const myArray = [];  
  
myArray.summary = "Empty array!";  
  
myArray;  
//=> [summary: "Empty array!"]
```

And we can modify and access those properties, too:

```
myArray["summary"] = "This array is totally empty.";  
  
myArray;  
//=> [summary: "This array is totally empty."]
```

```
myArray.summary;  
//=> "This array is totally empty."
```

In fact, *everything* we just learned how to do to **Object** s can also be done to **Array** s because **Array** s **are** **Object** s. Just special ones. To see the special stuff, let's **.push()** some values into our **Array** :

```
myArray.push(2, 3, 5, 7);  
//=> 4
```

```
myArray;  
//=> [2, 3, 5, 7, summary: "This array is totally empty."]
```

Cool, looks like everything's still in there. What's your guess about the **Array** 's **.length** ?

```
myArray.length;  
//=> 4
```

Huh, that's interesting. Surely our **summary** must be the first element in the **Array** , no? After all, we did add it before we **.push()** ed all those values in.

```
myArray[0];  
//=> 2
```

Hm, then maybe it's the last element?

```
myArray[myArray.length - 1];  
//=> 7
```

What the heck? Where is it?

You see, one of the 'special' features of an **Array** is that **its Array -style elements are stored separately from its Object -style properties**. The **.length** property of an **Array** describes how many items exist in its special list of elements. Its **Object** -style properties are not included

in that calculation.

This brings up an interesting question: if we add a new property to an **Array** that has a key of **0**, how does the JavaScript engine know whether it should be an **Object**-style property or an **Array**-style element?

```
const myArray = [];  
  
myArray[0] = "Will this be an `Object` property or an `Array` element?";  
//=> "Will this be an `Object` property or an `Array` element?"  
  
// Moment of truth...  
myArray.length;  
//=> 1  
  
myArray;  
//=> ["Will this be an `Object` property or an `Array` element?"]
```

So JavaScript used that assignment operation to add a new **Array**-style element. What happens if we enclose the integer in quotation marks, turning it into a string?

```
myArray["0"] = "What about this one?";  
//=> "What about this one?"  
  
myArray.length;  
//=> 1  
  
myArray;  
//=> ["What about this one?"]
```

This is hitting on a fundamental truth: **all keys in Objects and all indexes in Arrays are actually strings**. In `myArray[0]` we're using the integer **0**, but under the hood the JavaScript engine automatically converts that to the string **"0"**. When we access elements or properties of

an **Array**, the engine routes all integers and integers masquerading as strings (e.g., **'14'**, **"953"**, etc.) to the **Array**'s special list of elements, and it treats everything else as a simple **Object** property. For example:

```
const myArray = [2, 3, 5, 7];
```

```
myArray["1"] = "Hi";  
//=> "Hi"
```

```
myArray;  
//=> [2, "Hi", 5, 7]
```

```
myArray["01"] = "Ho";  
//=> "Ho"
```

```
myArray;  
//=> [2, "Hi", 5, 7, 01: "Ho"]
```

```
myArray[01];  
//=> "Hi"
```

```
myArray["01"];  
//=> "Ho"
```

After adding our weird **'01'** property, the **.length** property still returns **4**:

```
myArray.length;  
//=> 4
```

So it would stand to reason that **Object.keys()** would only return **'01'**, right?

```
Object.keys(myArray);  
//=> ["0", "1", "2", "3", "01"]
```

Unfortunately not. The reason why **Array** s have this behavior would take us deep inside the JavaScript source code, and it's frankly not that important. Just remember these simple guidelines, and you'll be just fine:

- **For accessing elements in an **Array** , always use integers.**
- **Be wary of setting **Object** -style properties on an **Array** .** There's rarely any reason to, and it's usually more trouble than it's worth.
- **Remember that all **Object** keys, including **Array** indexes, are strings.** This will really come into play when we learn how to iterate over **Object** s, so keep it in the back of your mind.

## Conclusion

In this and the previous lesson, we dug deep into **Object** s in JavaScript. We identified what an **Object** is and how to access values stored in it. We also covered how to add and remove properties and how to use some of JavaScript's convenience methods ( **Object.keys()** , **Object.values()** , and the spread operator). We also explored the relationship between **Object** s and **Array** s.

## Resources

- MDN
  - **Object basics** ➞ <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>
  - **Object.assign()** ➞ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)
  - **delete** ➞ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete>
  - **Spread Syntax in Object Literals** ➞ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_syntax#spread\\_in\\_object\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_object_literals)