3

JavaScript Multiple Values

The basic data types have been dealt with in the previous chapter. Now it's time to look at a slightly more complicated topic: arrays and objects. In the previous chapter, you saw variables that held just a single value. To allow for more complex programming, objects and arrays can contain multiple values.

You can look at objects as a collection of properties and methods. Properties can be thought of as variables. They can be simple data structures such as numbers and strings, but also other objects. Methods perform actions; they contain a certain number of lines of code that will be executed when the method gets called. We'll explain methods in more detail later in this book and focus on properties for now. An example of an object can be a real-life object, for example, a dog. It has properties, such as name, weight, color, and breed.

We will also discuss arrays. An array is a type of object, which allows you to store multiple values. They are a bit like lists. So, you could have an array of items to buy at the grocery store, which might contain the following values: apples, eggs, and bread. This list would take the form of a single variable, holding multiple values.

Along the way, we will cover the following topics:

- Arrays and their properties
- Array methods
- Multidimensional arrays
- Objects in JavaScript
- Working with objects and arrays

Let's start with arrays.

Note: exercise, project and self-check quiz answers can be found in the *Appendix*.

Arrays and their properties

Arrays are lists of values. These values can be of all data types and one array can even contain different data types. It is often very useful to store multiple values inside one variable; for example, a list of students, groceries, or test scores. Once you start writing scripts, you'll find yourself needing to write arrays very often; for example, when you want to keep track of all the user input, or when you want to have a list of options to present to the user.

Creating arrays

You might be convinced by now that arrays are great, so let's see how we can make them. There is actually a right way and a wrong way to do it. Here are both. Which one do you think is the right one?

```
arr1 = new Array("purple", "green", "yellow");
arr2 = ["black", "orange", "pink"];
```

If you guessed the second option, using square brackets, you are right. This is the best and most readable way to create a new array. On the other hand, the first option can do unexpected things. Look at both lines of code here. What do you think they will do?

```
arr3 = new Array(10);
arr4 = [10];
```

Probably, you sense that something is up here. They do not both create an array with one value, `10`. The second one, `arr4`, does. The first option creates an array with 10 undefined values. If we log the values like this:

```
console.log(arr3);
console.log(arr4);
```

Here is what it logs:

```
[ <10 empty items> ]
[ 10 ]
```

Thanks, JavaScript! That was very helpful. So, unless that is what you need to do, please use the square brackets!

As I already mentioned, we can have mixed arrays and arrays can hold any type of variable. The values of the array won't be converted to a single data type or anything like that. JavaScript simply stores all the variables with their own data type and value in the array:

```
let arr = ["hi there", 5, true];
console.log(typeof arr[0]);
console.log(typeof arr[1]);
console.log(typeof arr[2]);
```

This will output to the console:

```
string
number
boolean
```

The last array fun fact we will go over here is what happens if you define an array using `const`. You can change the values of a constant array, but you cannot change the array itself. Here is a piece of code to demonstrate:

```
const arr = ["hi there"];
arr[0] = "new value";
console.log(arr[0]);

arr = ["nope, now you are overwriting the array"];
```

The new value for the first element of the array is going fine, but you cannot assign a new value to the full array. Here is what it will output:

```
new value
TypeError: Assignment to constant variable.
```

Accessing elements

This beautiful array we just made would become much more powerful if we could access its elements. We can do this by referencing the array's index. This is something we did not specify when we created the array, and we did not need to either. JavaScript assigns an index to every value of the array. The first value is assigned the position of 0, the second 1, the third 2, and so on. If we want to call a specific value based on its position in the array, we can use the name of our array, add square brackets to the end, and put the index we want to access between the square brackets, like this:

```
cars = ["Toyota", "Renault", "Volkswagen"];
console.log(cars[0]);
```

This log statement will write `Toyota` to the console because we called for the position 0 of the array, which outputs the first value in the list.

```
console.log(cars[1]);
```

Calling index position 1 is giving us the second element in the array, which is `Renault`. This will be logged to the console.

```
console.log(cars[2]);
```

The third element in our array has index 2, so this one will log `Volkswagen`. What do you think will happen if we use a negative index or an index that is higher than the number of values we get?

```
console.log(cars[3]);
console.log(cars[-1]);
```

We didn't assign a value to the negative or non-existent index, so when we ask for it, the value is undefined. As such, the log output will be undefined. JavaScript does not throw an error because of this.

Overwriting elements

The elements in an array can be overwritten. This can be done by accessing a certain element using the index and assigning a new value:

```
cars[0] = "Tesla";
console.log(cars[0]);
```

The output of this log is `Tesla` because it has overwritten the old value, `Toyota`. If we output the whole array:

```
console.log(cars);
```

It will output the following:

```
[ 'Tesla', 'Renault', 'Volkswagen' ]
```

What happens if you try to overwrite an element that does not exist?

```
cars[3] = "Kia";
```

Or even a negative index?

```
cars[-1] = "Fiat";
```

Let's see what happens when we try to write the values to the console:

```
console.log(cars[3]);
console.log(cars[-1]);
```

And the output:

```
Kia
Fiat
```

Ha! They suddenly exist. How is that you may wonder? We will discuss this in the next section. For now, just remember that this is not the right way to add values to the array. We will discuss the right way when we explain arrays in the *Array methods* section.

Built-in length property

Arrays have a very useful built-in property: length. This will return the number of values that the array has:

```
colors = ["black", "orange", "pink"]
booleans = [true, false, false, true];
emptyArray = [];

console.log("Length of colors:", colors.length);
console.log("Length of booleans:", booleans.length);
console.log("Length of empty array:", emptyArray.length);
```

The first `console.log` call returns `3`, indicating that the colors array contains 3 values. The second one returns `4`, and the last one is an empty array with a length of `0`:

```
Length of colors: 3
Length of booleans: 4
Length of empty array: 0
```

So, be aware that the length is one higher than the maximum index because the index of the array starts at 0, but when determining the length, we look at the number of elements and there are four separate elements. This is why the maximum index is 3 when the length is 4. Hence, the positional value of the last element in the array will be one fewer than the total number of elements.

Take a moment and try to figure out how you can use the length to access the last element of the array:

```
lastElement = colors[colors.length - 1];
```

You get the highest index by subtracting 1 from the length because, as you know, arrays are zero-indexed. So, the positional value of the last element in the array will be one fewer than the total number of elements.

So, this might seem pretty straightforward. Remember the non-existent index position we called in the previous section? Let's see what happens in this example:

```
numbers = [12, 24, 36];
numbers[5] = 48;
console.log(numbers.length);
```

The length of the array is only counting the integer numbers starting from 0 up to the highest filled index. If there are elements in the middle of the sequence that do not have a value, they will still get counted. In this case, the length becomes 6. If we log the array, we can see why:

```
console.log("numbers", numbers);
```

The output will be as follows:

```
numbers [ 12, 24, 36, <2 empty items>, 48 ]
```

Because we added an element, 48, at index 5, it also created 2 elements at index positions 3 and 4 containing empty values. For now, let's have a look at array methods and find out the right way to add to an array.

Practice exercise 3.1

1. Create an array to use as your shopping list with 3 items: "Milk," "Bread," and "Apples."
2. Check your list length in the console.
3. Update "Bread" to "Bananas."
4. Output your entire list to the console.

Array methods

We have just seen the built-in length property. We also have a few built-in methods. Methods are functions on a certain object. Instead of holding a value, like properties, they perform actions. We will cover functions in-depth in *Chapter 6, Functions*. For now, all you need to know is that you can call methods and functions, and when you do, some code that is specified inside that function gets executed.

We just accidentally saw we could add elements using new indices. This is not the proper way to do it as it is easy to make mistakes and accidentally overwrite a certain value or skip a certain index. The right way is to do this with a special method. Similarly, we can also delete elements and sort the elements in the array.

Adding and replacing elements

We can add elements with the `push()` method:

```
favoriteFruits = ["grapefruit", "orange", "lemon"];
favoriteFruits.push("tangerine");
```

The value gets added to the end of the array. The `push` method returns the new length of the array, four in this case. You can store this length in a variable like this:

```
let lengthOfFavoriteFruits = favoriteFruits.push("lime");
```

The value 5 gets stored in the `lengthOfFavoriteFruits` variable. If we log our array, `favoriteFruits`, like this:

```
console.log(favoriteFruits);
```

Here is the new array:

```
[ 'grapefruit', 'orange', 'lemon', 'tangerine', 'lime' ]
```

This was easy right? But what if you would want to add elements at a certain index? You can use the `splice()` method. This one is slightly more difficult:

```
let arrOfShapes = ["circle", "triangle", "rectangle", "pentagon"];
arrOfShapes.splice(2, 0, "square", "trapezoid");
console.log(arrOfShapes);
```

After this, the output containing the array is as follows:

```
[
  'circle',
  'triangle',
  'square',
  'trapezoid',
  'rectangle',
  'pentagon'
]
```

First, let's point out the different layouts of this output. This might depend on the interpreter you are using, but at some point, it will decide it is too long for a single line and apply an automatic format to the array to make it more readable. It doesn't change the value of the array; it is just a different representation of the same values were they to be on a single line.

As you can see, the square and trapezoid get inserted on index 2. The rest of the array is shifting to the right. The `splice()` method takes multiple parameters. The first parameter, 2 in our case, is the index of the array on which we want to start inserting. The second parameter, 0 in our case, is the number of elements we want to delete starting at our previously defined starting index. The parameters after these first two, `square` and `trapezoid` in our case, are whatever should be inserted starting at the start index.

So, had we said this instead:

```
arrOfShapes.splice(2, 2, "square", "trapezoid");
console.log(arrOfShapes);
```

It would have replaced the elements `rectangle` and `pentagon` and added `square` and `trapezoid` in their place, as follows:

```
[ 'circle', 'triangle', 'square', 'trapezoid' ]
```

If you were to increase the second parameter to a number higher than our array, it would not affect the result as JavaScript would simply stop as soon as it runs out of values to delete. Try the following code:

```
arrOfShapes.splice(2, 12, "square", "trapezoid");
console.log(arrOfShapes);
```

This would also have had this output:

```
[ 'circle', 'triangle', 'square', 'trapezoid' ]
```

You can also add another array to your array. This can be done with the `concat()` method. This way, you can create a new array that consists of a concatenation of both arrays. The elements of the first array will be first, and the elements of the argument of `concat()` will be concatenated to the end:

```
let arr5 = [1, 2, 3];
let arr6 = [4, 5, 6];
let arr7 = arr5.concat(arr6);
console.log(arr7);
```

And here is the output:

```
[ 1, 2, 3, 4, 5, 6 ]
```

This `concat()` method can do even more! We can use it to add values as well. We can add a single value, or we can comma-separate multiple values:

```
let arr8 = arr7.concat(7, 8, 9);
console.log(arr8);
```

The new value of the array will be as follows:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Deleting elements

There are several ways in which you can delete elements from an array. Removing the last element is done with `pop()`:

```
arr8.pop();
```

Logging the array after executing `pop()` results in this:

```
[ 1, 2, 3, 4,  5, 6, 7, 8 ]
```

Deleting the first element can be done with `shift()`. This causes all other indices to be reduced by one:

```
arr8.shift();
```

The new array will be:

```
[ 2, 3, 4, 5,  6, 7, 8 ]
```

Remember `splice()`? This is a very special method because we can also use it to delete values. We specify the index from where we want to start deleting, and then the number of elements we want to delete.

```
arr8.splice(1, 3);
```

After this, the value of the array is as follows:

```
[ 2, 6, 7, 8 ]
```

As you can see, 3 elements starting from the second positional index have been deleted. The values 3, 4, and 5 are gone. If you do not wish to change any of the later indices, you can also use the operator delete. This is not a method, but you can use it to change the value of a certain position of the array to undefined:

```
delete arr8[0];
```

The array then becomes:

```
[ <1 empty item>, 6, 7, 8 ]
```

This is useful when you are relying on index or length for some reason. For example, if you are keeping user input, and you want to determine the number of user inputs based on an array that the user is pushing to, deleting would decrease the number of inputs, whereas that might not be what you would want.

Finding elements

If you want to check whether a value is present in an array, you can use the `find()` method. What will go in the `find()` method is somewhat different. It is actually a function. This function will be executed on every element in the array until it finds a match, and if it does not, then it will return undefined.

Do not worry if this is too difficult for now; it will become clear soon enough. We are writing the function in two different ways in the following code snippet. They are actually doing the same, except that the first one is checking for an element being equal to 6, and the second for an element being equal to 10:

```
arr8 = [ 2, 6, 7, 8 ];
let findValue = arr8.find(function(e) { return e === 6});
let findValue2 = arr8.find(e => e === 10);
console.log(findValue, findValue2);
```

The log statement will log `6` and `undefined` because it can find an element that matches for `6`, but not one that matches for `10`.

A function can take a certain input. In this case, it takes the element of the array as an input. When the element of the array equals 6 (`findValue`) or 10 (`findValue2`), it returns the element. In *Chapter 6*, *Functions*, we will cover functions in much more detail. It is a lot for a beginner to take in, so you can review this a bit later if it is unclear for now.

Often, you do not only want to find the element, but you want to know what position it is on. This can be done with the `indexOf()` method. This method returns the index on which the value is found. If a value occurs in an array more than once, it will return the first occurrence. If the value is not found, it will return `-1`:

```
arr8 = [ 2, 6, 7, 8 ];
let findIndex = arr8.indexOf(6);
let findIndex2 = arr8.indexOf(10);
console.log(findIndex, findIndex2);
```

So, the first one will return 1 since that is the index position of 6 in the array. The second one will return -1 because the array does not contain 10.

If you want to find the next occurrence of the specified number, you can add a second argument to `indexOf()`, specifying from which position it should start searching:

```
arr8 = [ 2, 6, 7, 8 ];
let findIndex3 = arr8.indexOf(6, 2);
```

In this case, the value of `findIndex3` will be -1, because 6 cannot be found starting from index 2.

The last occurrence can also be found. This is done with the `lastIndexOf()` method:

```
let animals = ["dog", "horse", "cat", "platypus", "dog"];
let lastDog = animals.lastIndexOf("dog");
```

The value of `lastDog` will be 4 because that is the last occurrence of `dog` in the array.

Sorting

There is also a built-in method for sorting arrays. It sorts numbers from small to high and strings A-Z. You can call `sort()` on an array and the order of the values of the array will change to a sorted order:

```
let names = ["James", "Alicia", "Fatiha", "Maria", "Bert"];
names.sort();
```

The value of names after sorting is as follows:

```
[ 'Alicia', 'Bert', 'Fatiha', 'James', 'Maria' ]
```

As you can see, the array is now sorted alphabetically. For numbers, it is sorting them in ascending order, as you can see in the following code snippet:

```
let ages = [18, 72, 33, 56, 40];
ages.sort();
```

After executing this `sort()` method, the value of ages is:

```
[ 18, 33, 40, 56, 72 ]
```

Reversing

The elements of the array can be reversed by calling the built-in method, `reverse()`, on an array. It puts the last element first, and the first element last. It does not matter whether the array is sorted or not; it just reverses the order.

The value of names before reversing is as follows:

```
[ 'Alicia', 'Bert', 'Fatiha', 'James', 'Maria' ]
```

Now we are going to call the `reverse()` method:

```
names.reverse();
```

The new order will be:

```
[ 'Maria', 'James', 'Fatiha', 'Bert', 'Alicia' ]
```

Practice exercise 3.2

1. Create an empty array to use as a shopping list.
2. Add `Milk`, `Bread`, and `Apples` to your list.
3. Update "`Bread`" with `Bananas` and `Eggs`.
4. Remove the last item from the array and output it into the console.
5. Sort the list alphabetically.
6. Find and output the index value of `Milk`.
7. After `Bananas`, add `Carrots` and `Lettuce`.
8. Create a new list containing `Juice` and `Pop`.
9. Combine both lists, adding the new list twice to the end of the first list.
10. Get the last index value of `Pop` and output it to the console.
11. Your final list should look like this:

```
["Bananas", "Carrots", "Lettuce", "Eggs", "Milk", "Juice",
"Pop", "Juice", "Pop"]
```

Multidimensional arrays

Earlier, we established already that arrays can contain any data type. This means that arrays can also contain other arrays (which, in turn, can contain… other arrays!). This is called a multidimensional array. It sounds complicated, but it is just an array of arrays: a list of lists:

```
let someValues1 = [1, 2, 3];
let someValues2 = [4, 5, 6];
let someValues3 = [7, 8, 9];

let arrOfArrays = [someValues1, someValues2, someValues3];
```

So, we can create an array of already existing arrays. This is called a two-dimensional array. We can write it like this:

```
let arrOfArrays2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

If you want to access elements of the inner arrays, you will have to specify an index twice:

```
let value1 = arrOfArrays[0][1];
```

The statement will grab the first array because it has an index position of `0`. From this first array, it will take the second value, because it has an index position of `1`. Then it stores this value in `value1`. That means the value of `value1` will be 2. Can you figure out what the value of the next one will be?

```
let value2 = arrOfArrays[2][2];
```

It takes the third array, and from this third array, it takes the third value. Thus, 9 will be stored in `value2`. And it does not stop here; it can go many levels deep. Let's show that by creating an array of our array of arrays. We are simply going to store this array three times in another array:

```
arrOfArraysOfArrays = [arrOfArrays, arrOfArrays, arrOfArrays];
```

This is what the array looks like in terms of values:

```
[
  [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ],
  [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ],
  [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
]
```

Let's get the middle element of this array, which is the value 5, belonging to the second array of arrays. It is done like this:

```
let middleValue = arrOfArraysOfArrays[1][1][1];
```

The first step is to get the second array of arrays, so index 1. Then we need to get the second array of this one, which again is index 1. Now we reach the level of the values, and we need the second value, so again we use index 1. This is useful in many situations, for example, when you want to work with matrices.

Practice exercise 3.3

1. Create an array containing three values: 1, 2, and 3.
2. Nest the original array into a new array three times.
3. Output the value 2 from one of the arrays into the console.

Objects in JavaScript

Now it is time to have a look at another complex data structure that can contain more than one value: objects! Objects are very useful and can be used to describe real-life objects as well as more complex abstract concepts that allow for more flexibility in your code.

Secretly, you have just been introduced to objects already, because arrays are a very special type of object. Arrays are objects with indexed properties. All the other objects, and also the objects we will see here, are objects with named properties. This means that instead of an automatically generated index number, we will give it a custom descriptive name.

As we can tell from the following code, arrays are defined by JavaScript as being of the object type:

```
let arr = [0, 1, 2];
console.log(typeof arr);
```

The output of the preceding code is as follows:

```
Object
```

Objects are not too dissimilar to real-world objects. They *have* properties and they *can perform* actions, methods. Here, we will only deal with the properties. We will cover methods in *Chapter 7*, *Classes*, after we have seen functions. An object is a chance to group multiple variables into one. This is done with curly brackets: { and }. Let's have a look at this object of a dog here:

```
let dog = { dogName: "JavaScript",
            weight: 2.4,
            color: "brown",
            breed: "chihuahua",
            age: 3,
            burglarBiter: true
          };
```

We created a variable, `dog`, and we gave this an object as a value. We can recognize that this is an object by seeing the { and }. In between the curly braces, we see a bunch of properties and their values.

If you have ever wondered whether something should be a property, just try the following template sentence in your head:

objectname* has a(n) *property name

For example, a dog has a name, a dog has a color, and a dog has a weight. This is slightly different for the Boolean properties, for which you can use "is" or "is not" instead of "has".

We can access the properties of this object in a very similar way as we would with the array. This time, we are not using the index number, but the name of the property, to get the value:

```
let dogColor1 = dog["color"];
```

There is another way to do this. Instead of the square brackets, the property name can also be added to the object name with a dot in between:

```
let dogColor2 = dog.color;
```

This might look familiar. Do you remember how we got the length of an array with the built-in property length? Yes—the same way! The difference between properties and methods is the lack of parentheses for properties.

Updating objects

We can change the value of the properties of the objects. Again, this is similar to an array because an array is an object as well, but for properties, we have two options:

```
dog["color"] = "blue";
dog.weight = 2.3;
```

This has changed the properties of our chihuahua JavaScript. The color gets updated to blue and it has lost a little bit of weight since the new weight is 0.1 lower. So if we log our dog:

```
console.log(dog);
```

We will get the following:

```
{
  dogName: 'JavaScript',
  weight: 2.3,
  color: 'blue',
```

```
  breed: 'chihuahua',
  age: 3,
  burglarBiter: true
}
```

It's useful to note that if we change the data type of one of our properties, for example:

```
dog["age"] = "three";
```

This is not a problem. JavaScript will just change the whole value and data type to the new situation.

Another element to note is that we are now using the literal string values to refer to the object's properties, but we can also work with variables to achieve this. So, for example:

```
let variable = "age";
console.log(dog[variable]);
```

This will still output `three`, as we just changed the value of age to three. If we change the value of the variable to another dog property, we will be accessing another property, like this:

```
variable = "breed";
console.log(dog[variable]);
```

This will print `chihuahua`. And when we update the value accessing this way, it is the exact same as when we would have accessed it with the literal string:

```
dog[variable] = "dachshund";
console.log(dog["breed"]);
```

So, this will log `dachshund` to the console.

Practice exercise 3.4

1. Create a new `myCar` object for a car. Add some properties, including, but not limited to, `make` and `model`, and values for a typical car or your car. Feel free to use booleans, strings, or numbers.

2. Create a variable that can hold the string value `color`. This variable containing a string value color can now be used to reference the property name within `myCar`. Then, use the variable within the square bracket notation to assign a new value to the color property in `myCar`.
3. Use that same variable and assign a new property string value to it, such as `forSale`. Use the bracket notation once again to assign a new value to the `forSale` property to indicate whether the car is available for purchase.
4. Output `make` and `model` into the console.
5. Output the value of `forSale` into the console.

Working with objects and arrays

When working with objects and arrays, you will see these often combined. In the last section of this chapter, we will deal with combining objects and arrays, and also objects inside objects.

Objects in objects

Let's say we want to have an object for a company. This company will have an address. And an address is another object. If we give our company an address, we are using an object inside an object:

```
let company = { companyName: "Healthy Candy",
                activity: "food manufacturing",
                address: {
                  street: "2nd street",
                  number: "123",
                  zipcode: "33116",
                  city: "Miami",
                  state: "Florida"
                },
                yearOfEstablishment: 2021
              };
```

As you can see, our company object has an address object with values. This can go very many levels deep if necessary.

To access or modify one of the properties of the address here, we can use two approaches:

```
company.address.zipcode = "33117";
company["address"]["number"] = "100";
```

As you can see, this is very similar to the array. We first need to select the address and then do the same thing to access the property we want to change.

Arrays in objects

Our company might have a range of activities instead of one. We can simply replace the activity from our previous sample with an array:

```
company = { companyName: "Healthy Candy",
                activities: ["food manufacturing",
"improving kids' health", "manufacturing toys"],
                address: {
                  street: "2nd street",
                  number: "123",
                  zipcode: "33116",
                  city: "Miami",
                  state: "Florida"
                },
                yearOfEstablishment: 2021
              };
```

We have now used an array in our company object. You can simply use an array with the square braces after the property. Retrieving the individual values is very similar. The second value of the activities array can be fetched using this statement:

```
let activity = company.activities[1];
```

Here, we call the object we're interested in, `company`, then the relevant array, `activities`, with reference to the index position of the variable we're looking for within the array, which is `1`.

Objects in arrays

It is very possible that instead of one address, our company has a list of addresses. We can accomplish this by creating an array of address objects. In this case, we will create an array of two:

```
let addresses = [{
    street: "2nd street",
    number: "123",
    zipcode: "33116",
    city: "Miami",
    state: "Florida"
  },
  {
    street: "1st West avenue",
    number: "5",
    zipcode: "75001",
    city: "Addison",
    state: "Texas"
  }];
```

So, the arrays can be recognized by the square brackets and the objects by the curly brackets. The street name of the first object can be fetched using this statement:

```
let streetName = addresses[0].street;
```

Here, we call the array we're interested in, `addresses`, with reference to the index position of the object we're looking for within the array, `0`, and then the required variable from within the object, which is `street`. This may seem complicated, but you may notice that this simply reverses the syntax required to retrieve a variable from an array inside an object from the previous section. It's worth practicing calling variables from nested arrays and objects until you're comfortable with it!

Objects in arrays in objects

Just to show that this can go as many levels as we would need, we are going to give our company object an array of address objects. So, let's add this array of address objects to our company object. This way, our company has an array of addresses:

```
  company = { companyName: "Healthy Candy",
                    activities: [ "food manufacturing",
"improving kids' health",
```

```
"manufacturing toys"],
                    address: [{
                      street: "2nd street",
                      number: "123",
                      zipcode: "33116",
                      city: "Miami",
                      state: "Florida"
                    },
                    {
                      street: "1st West avenue",
                      number: "5",
                      zipcode: "75001",
                      city: "Addison",
                      state: "Texas"
                    }],
                    yearOfEstablishment: 2021
                  };
```

To access elements of increasingly nested objects and arrays, we simply extend the same logic you have seen in the previous sections. To access the street name of Healthy Candy's first address, we can use the following code:

```
let streetName = company.address[0].street;
```

As you see, we can stack object and array element requests indefinitely.

We will not make it any more complicated than this for now. Whenever you need a list of something, you will be using an array. Whenever you want to represent something with properties that have descriptive names, it is better to use an object. Just remember that object properties can be of any type.

Practice exercise 3.5

1. Create an object named `people` that contains an empty array that is called `friends`.
2. Create three variables, each containing an object, that contain one of your friend's first names, last names, and an ID value.
3. Add the three friends to the `friend` array.
4. Output it to the console.

Chapter projects

Manipulating an array

Take the following array:

```
const theList = ['Laurence', 'Svekis', true, 35, null, undefined,
{test: 'one', score: 55}, ['one', 'two']];
```

Manipulate your array using various methods, such as `pop()`, `push()`, `shift()`, and `unshift()`, and transform it into the following:

```
["FIRST", "Svekis", "MIDDLE", "hello World", "LAST"]
```

You can take the following steps, or adopt your own approach:

- Remove the first item and the last item.
- Add `FIRST` to the start of the array.
- Assign `hello World` to the fourth item value.
- Assign `MIDDLE` to the third index value.
- Add `LAST` to the last position in the array.
- Output it to the console.

Company product catalog

In this project, you will implement a data structure for a product catalog and create queries to retrieve data.

1. Create an array to hold an inventory of store items.
2. Create three items, each having the properties of name, model, cost, and quantity.
3. Add all three objects to the main array using an array method, and then log the inventory array to the console.
4. Access the quantity element of your third item, and log it to the console. Experiment by adding and accessing more elements within your data structure.

Self-check quiz

1. Can you use `const` and update values within an array?
2. Which property in an array gives the number of items contained in the array?
3. What is the output in the console?

```
const myArr1 = [1,3,5,6,8,9,15];
console.log(myArr1.indexOf(0));
console.log(myArr1.indexOf(3));
```

4. How do you replace the second element in an array `myArr = [1,3,5,6,8,9,15]` with the value 4?
5. What is the output in the console?

```
const myArr2 = [];
myArr2[10] = 'test'
console.log(myArr2);
console.log(myArr2[2]);
```

6. What is the output in the console?

```
const myArr3 = [3,6,8,9,3,55,553,434];
myArr3.sort();
myArr3.length = 0;
console.log(myArr3[0]);
```

Summary

So, in this chapter, we have seen arrays and objects. Arrays are a list of values. These could be values of the same type, but also values of different types. Every element of the array gets an index. The index of the first element is 0. We can access the elements of the array using this index. We can also use this index to change and delete the element.

We then saw that it is also possible to have arrays containing other arrays; these are multidimensional arrays. To access the elements of a multidimensional array, you would need to use as many indices as you have nested arrays.

Then, we covered objects and learned that arrays are a special kind of object. Objects contain properties and methods. We looked at the properties of objects and saw that these properties are given a name and can be accessed and modified using this name.

We ended this module by looking at how arrays can contain objects, and how objects can contain arrays and more. This enables us to create complex object structures, which will be of great use in designing real-life applications.