HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Web Programming
JavaScript – node.js - react

Lecture 1-2:
JavaScript Update ES6+
19.10.

Stefan Noll
Manuel Fehrenbach
Winter Semester 22/23

Spread operators

ES6 introduced a new operator called the spread operator. The spread operator is also denoted by the prefix three dots (...). However, the spread operator is completely different from the rest parameters that we learnt in the last section. The most literal translation of the task done by the spread operator is to spread out the elements of an iterable object like an array.

To understand spread operators better, let's check out the different functionalities of the spread operator:

- **Insert an array/object in another array/object**

  The spread operator allows you to insert an array into another array initialisation. Let's understand this with an example:

```
> let initialCharacters = ['Rachel', 'Ross', 'Monica', 'Joey'];
  let allCharacters = [...initialCharacters, 'Phoebe', 'Chandler'];
  let allFriends = ['Phoebe', 'Chandler', ...initialCharacters];

  console.log("All characters:    ", allCharacters);
  console.log("Changing the order of inserting initial characters:   ",
  allFriends);
  All characters:                                                VM10447:5
  ▶ (6) ["Rachel", "Ross", "Monica", "Joey", "Phoebe", "Chandler"]
  Changing the order of inserting initial characters:            VM10447:6
  ▶ (6) ["Phoebe", "Chandler", "Rachel", "Ross", "Monica", "Joey"]
```

Figure 6.28: *Inserting an array into another array initialisation*

In the preceding screenshot, we have an array of initial characters with four values on the first line. In the next line, we are creating another array by the name **`allCharacters`** where we want to add the list of characters listed in the array **`initialCharacters`** along with two new characters. To achieve this, we simply use the spread operator to insert the **`initialCharacters`** array at a particular position. As you can see in the second and the third lines of *figure 6.28*, the spread operator will insert the entire array only at the position where you use the spread operator.

As shown in the following screenshot, you can also insert an object into the initialisation of another object using the spread operator:

```
> let accomodation = {
      country: 'India',
      state: 'Maharashtra',
      city: 'Mumbai',
      postalCode: 400058
  }

  let person = {
      name: 'John Doe',
      age: 27,
      ...accomodation
  }

  console.log("Person:    ", person);
  Person:                                                    VM8877:14
  ▼{name: "John Doe", age: 27, country: "India", state: "Maharashtra", city:
  "Mumbai", …} i
      age: 27
      city: "Mumbai"
      country: "India"
      name: "John Doe"
      postalCode: 400058
      state: "Maharashtra"
    ▶ __proto__: Object
```

Figure 6.29: *Inserting an object into another object initialisation*

- **Concatenate arrays/objects**

The spread operator can also be used to concatenate two or more arrays or objects. As shown in the following screenshot, to concatenate multiple arrays, you need to insert the name of the arrays with the spread operator prefix (...), in the order you want.

Figure 6.30: *Concatenating two arrays using the spread operator*

Similarly, you can concatenate multiple objects using the same technique of the spread operators, as shown in the following screenshot:

Figure 6.31: *Concatenating two objects using the spread operator*

- **Copying arrays/objects**

In the previous chapter, we had learnt that the arrays and the object follow the concept of '*copy by reference*'. As shown in the following screenshot, it basically means that when you copy an object, its value is not copied but a reference to the object is copied. Therefore, when you change any one of the object, the value of the other object also changes.

```
> let person1 = { name: 'John Doe', age: 27 };
  let person2 = person1;

  console.log("Person 1:    ", person1);
  console.log("Person 2:    ", person2);

  console.log("******* Changing the value of person1 *******");

  person1.name = 'Jane Dias';

  console.log("Person 1 after value change:    ", person1);
  console.log("Person 2 after value change:    ", person2);
Person 1:     ▶ {name: "John Doe", age: 27}                    VM14093:4
Person 2:     ▶ {name: "John Doe", age: 27}                    VM14093:5
******* Changing the value of person1 *******                  VM14093:7
Person 1 after value change:     ▶ {name: "Jane Dias", age: 27}  VM14093:11
Person 2 after value change:     ▶ {name: "Jane Dias", age: 27}  VM14093:12
```

Figure 6.32: *Objects follow the concept of copy by reference*

```
> let person1 = { name: 'John Doe', age: 27 };
  let person2 = person1;

  console.log("Person 1:    ", person1);
  console.log("Person 2:    ", person2);

  console.log("******* Changing the value of person1 *******");

  person1.name = 'Jane Dias';

  console.log("Person 1 after value change:    ", person1);
  console.log("Person 2 after value change:    ", person2);
Person 1:     ▶ {name: "John Doe", age: 27}                    VM14093:4
Person 2:     ▶ {name: "John Doe", age: 27}                    VM14093:5
******* Changing the value of person1 *******                  VM14093:7
Person 1 after value change:     ▶ {name: "Jane Dias", age: 27}  VM14093:11
Person 2 after value change:     ▶ {name: "Jane Dias", age: 27}  VM14093:12
```

Figure 6.32: *Objects follow the concept of copy by reference*

This could be particularly disastrous if you want to make a change to only one of the object but the value of both the objects change. Think of the example in the preceding screenshot. In a practical analogy, the objects person1 and person2 are two different individuals. If we change the age of one of the persons, it's not correct that the age of the other person should increase. Or say, if we add the nationality of person2 as Spanish, it does not conclude that the person1 is also Spanish.

To solve this dilemma, we can use the spread operator. When we use the spread operator to copy an object, it copies the content of the object and not its reference.

As shown in the following screenshot, to use a spread operator to copy an object, you open a pair of curly braces. Inside the curly braces, you use the spread operator notation (...) and type the name of the object you want to copy. Similarly, if you want to copy an array, you follow the same steps, except that instead of the curly braces, you use the array notation, that is, square brackets ([]).

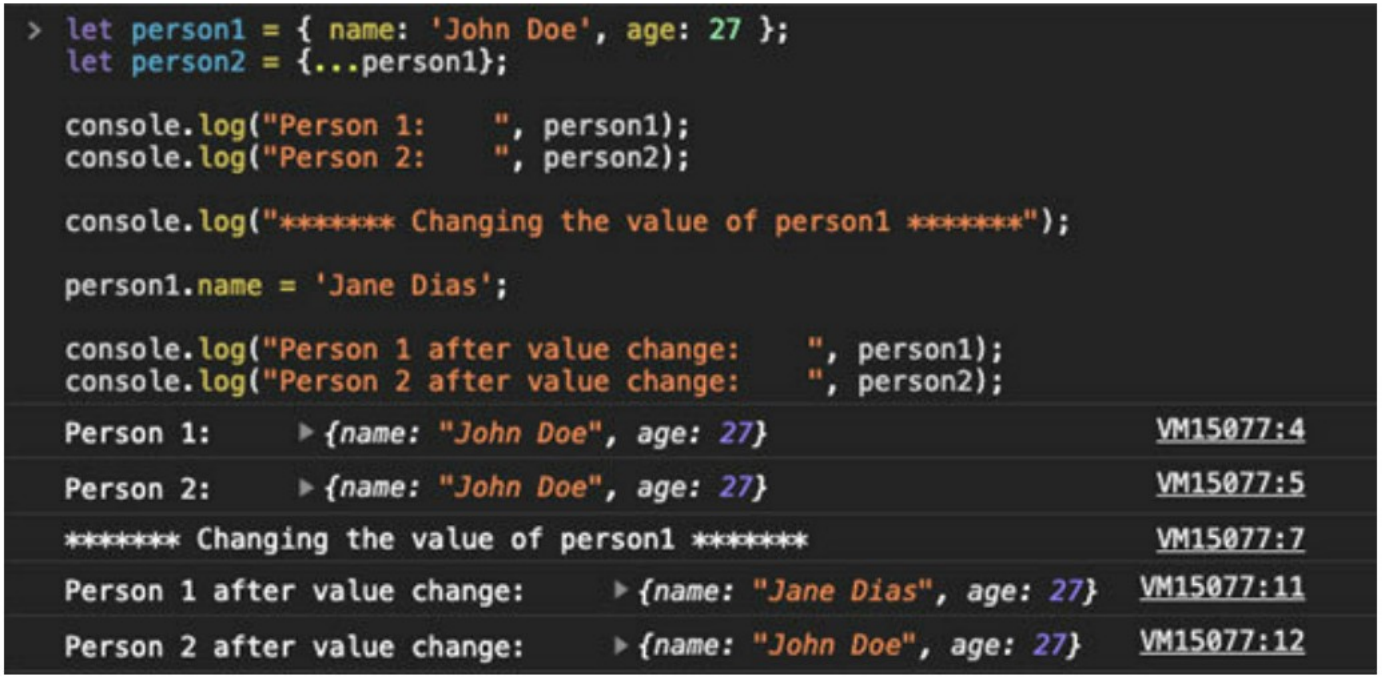

Figure 6.33: *Using spread operator to copy value of an object*

Template literals

So far, whenever we want to use variables in a string, we use single quotes (') or double quotes (") to create the text of the string. To insert variables in this string, we have to end the quotes and then insert the variables using the plus operator (+). This is called as a string literal.

As shown in the following *code example 6.5*, we face a lot of issues while creating string literals using quotes. From breaking the string at multiple points to using escape characters (\) to ensure the string does not crash, writing a string literal is seldom easy.

```
1. let name = "John Doe";
2. let age = 27;
3. let nationality = "British";


4.
5. let string = "Today's first candidate is "
   + name + ". He is " + age + " years old.
   His nationality is " + nationality + ". His
   favorite quote is \"If not now, then when?
   \"";
```

Code 6.5: *String literals*

To solve this, ES6 introduced template literals. Template literals help us to work with strings in a simpler, cleaner, and safer way. To use template literals, we wrap the string in backticks (`). Whenever we want to insert a variable in the template literal, we use a special syntax - **`${variableName}`**. Let's understand this with an example:

```
1. let name = "John Doe";
2. let age = 27;
3. let nationality = "British";


4.
5. let string = `Today's first candidate is ${name}.
6. He is ${age} years old. His nationality is ${nationality}.
7. His favorite quote is "If not now, then when?"`;
```

Code 6.6: *Template literals*

```
let name = "John Doe";
let age = 27;
let nationality = "British";

let string = `Today's first candidate is ${name}.
He is ${age} years old. His nationality is ${nationality}.
His favorite quote is "If not now, then when?"`;
```

Code 6.6: *Template literals*

```
let name = "John Doe";
let age = 27;
let nationality = "British";

let string = "Today's first candidate is " + name + ". He is " + age + " years old. His nationality is " + nationality + ". His favorite quote is \"If not now, then when?\"";
```

Code 6.5: *String literals*

As shown in the *code example 6.6*, we solve the following three problems with the help of template literals:

- **String formatting**: We can substitute variables easily compared to adding it using the traditional string literals.
- **HTML Escaping**: In the *code example 6.5*, since we were using double quotes, we had to use an escape character to ensure the double quote for the string is not considered as the end of the string. Template literals save us a lot of trouble here by avoiding the escaping of characters.
- **Multi-line statements**: In the *code example 6.5*, we will need to use the escape character (\n) to enter a new line. In template literals, it is as simple as entering the text on a new line.

Arrow functions

In *Chapter 3: Functions*, earlier in this book, we learned a great deal about functions. In 2015, ES6 introduced a much shorter syntax for writing the functions. This new syntax for functions are called as arrow functions. A funny name for a function, right? The arrow functions are called so because the most important syntax in an arrow function is the fat arrow notation (=>).

Arrow functions provide two significant advantages over traditional approach to writing functions:

- Shorter syntax
- Removes the binding of the **`this`** keyword

Shorter syntax

Let's try to understand the arrow functions with examples:

```
1. var add = function(number1, number2) {
2.   return number1 + number2;
3. }
```

Code 6.7: *Traditional method of writing the functions*

As shown in the preceding *code example 6.7*, we have created an anonymous function add using the traditional method of writing functions. ES6 reduces the same function to a much shorter syntax, as shown in the following *code example 6.8*:

```
1. let add = (number1, number2) => number1 +
   number2;
```

Code 6.8: *ES6 method of writing the functions*

```
1. let add = (number1, number2) => number1 +
   number2;
```

Code 6.8: *ES6 method of writing the functions*

So what exactly happened here? Let's understand it word by word:

1. The first thing that an arrow function does is removing the dependency on the keyword function.
2. Next, after you add the function parameters (if any), you add the fat arrow notation of ES6.
3. Now comes the interesting part. If the function is to return some value and the code that needs to be executed is only one line, you can completely discard the curly braces and the keyword return.

As you can see in the following screenshot, the function performs in the exact same manner:

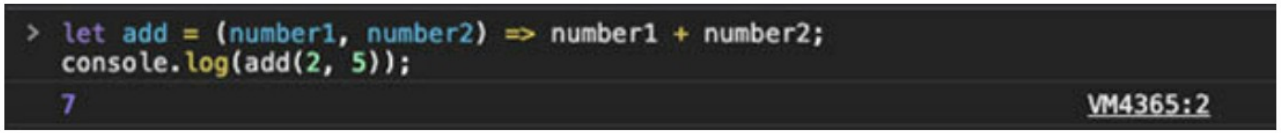

Figure 6.34: *The arrow function variant of code example 6.8*

If your function has no parameters, you need to provide empty parentheses. If your function has only one parameter, then you can discard the parentheses as well. This is depicted in the following *code example 6.9*:

```
1. let greeting = () => `Good morning, user!`;
2. let greeting = name => `Good morning, $
   {name}!`;


3.
4. greeting();      //Good morning, user!
5. greeting('John');      //Good morning, John!
```

Code 6.9: *ES6 method of writing the function with zero or one parameter*

The this keyword in ES6 functions

The arrow functions change the scope of the **`this`** keyword in functions. This change makes function execution much more intuitive.

Traditionally, when we use the **`this`** keyword, it refers to the owner of the function where it is executed or the window object. In the following screenshot, we are printing two statements on the console. The first statement is printed on the very first line inside the **`getFullName`** method. The second statement is inside an anonymous function stored in the variable print. This anonymous function is a part of the **`getFullName`** method.

Figure 6.35: *The "this" keyword in tradtional functions*

Any idea why the inner statement is returning undefined?

```
> let person = {
      firstName: 'John',
      lastName: 'Doe',
      getFullName: function() {
          console.log(`Outer: ${this.firstName} ${this.lastName}`);
          let print = function() {
              console.log(`Inner: ${this.firstName} ${this.lastName}`);
          };
          print();
      }
  }
  person.getFullName();
  Outer: John Doe                                    VM7472:5
  Inner: undefined undefined                         VM7472:7
```

Figure 6.35: *The "this" keyword in tradtional functions*

Any idea why the inner statement is returning undefined?

We have learned that the **`this`** keyword refers to the owner of the function in which it is used. In the case of the outer statement, the **`this`** keyword belongs to the method **`getFullName`**, and the owner of the method is the object person. Fair to say, since the person object has both the **`firstName`** and the **`lastName`** properties, we get the value of the outer statement as John Doe.

However, in the case of the inner statement, the **`this`** keyword belongs to the function inside the variable print. The owner of the anonymous function print is the method **`getFullName`** and from *figure 6.35*, we can see that the v method does not have the **`firstName`** and the **`lastName`** properties.

The arrow function solves this complexity by removing the bindings of the **`this`** keyword. Let's understand that with an example:

***Figure 6.36**: The "this" keyword in arrow functions*

In the preceding screenshot, we have used the arrow function for the inner statement. As we can see, the output of the inner statement now is John Doe, as well. So what happened here?

The arrow functions do not have their own **this**. So, the **`this`** value of the enclosing or the parent scope is used. In *figure 6.36*, since the anonymous function print does not have a this keyword, it looks for the this keyword in the scope of the **`getFullName`** method. And as we have seen earlier, since the **`getFullName`** method belongs to the person object, it derives its value from there.

New array methods – map, filter, and reduce

In *Chapter 4: Arrays*, earlier in this book, we learned about arrays. In that chapter, we learned that one of the ways to access every element of an array is to use a for loop. ES6 introduced three new ways to manipulate an array, which are far more simpler and effective than the for loop. We will go through each of these methods one-by-one.

map()

The **`map()`** method calls a function for every element in the array. The results of every function for every element is then populated into a new array by the **`map()`** method. The **`map()`** method does not change the original array.

Let's understand the **`map()`** method with a simple example. If I give you an array of numbers and ask you to double the value of all the elements inside the array, how will you do it?

Of the many methods that you can employ, the best method will be the following:

```
1. var numbers = [2, 4, 6];
2. var double = [];


3.
4. for(var i=0; i<numbers.length; i++) {
5.   let value = numbers[i]*2;
6.   double.push(value);
7. }
8. console.log("Double:    ", double);  //Dou-
   ble:    [4, 8, 12];
```

***Code 6.10**: Traditional method to double values in an array*

ES6 simplifies this to a greater extent by the **`map()`** method.

```
1. let numbers = [2, 4, 6];
2. let double = numbers.map(number => number *
   2);


3.
4. console.log("Double:    ", double);       //
   Double:    [4, 8, 12]
```

***Code 6.11**: ES6 method to double values in an array*

```
1. let numbers = [2, 4, 6];
2. let double = numbers.map(number => number *
   2);


3.
4. console.log("Double:   ", double);      //
   Double:   [4, 8, 12]
```

***Code 6.11**: ES6 method to double values in an array*

As you can see in the preceding *code example 6.11*, we have used the arrow functions in the map method. We call the map method on the array numbers. The map method passes through every element of the array one by one, performs some calculation (execution) on the element at that point, and returns the calculated (executed) value at the exact position in a new array.

```
1. let numbers = [2, 4, 6];
2. let double = numbers.map(number => number *
   2);

3.
4. console.log("Double:   ", double);      //
   Double:   [4, 8, 12]
```

***Code 6.11**: ES6 method to double values in an array*

In the *code example 6.11*, the variable number refers to every element in the array numbers. Going by the functionality of the arrow functions, every element of the array defined by the variable number is multiplied by two and that value is returned at the same position in a new array.

```
1. let numbers = [2, 4, 6];
2. let double = numbers.map(number => number *
   2);
3.
4. console.log("Double:   ", double);       //
   Double:   [4, 8, 12]
```

***Code 6.11**: ES6 method to double values in an array*

You can imagine it visually as

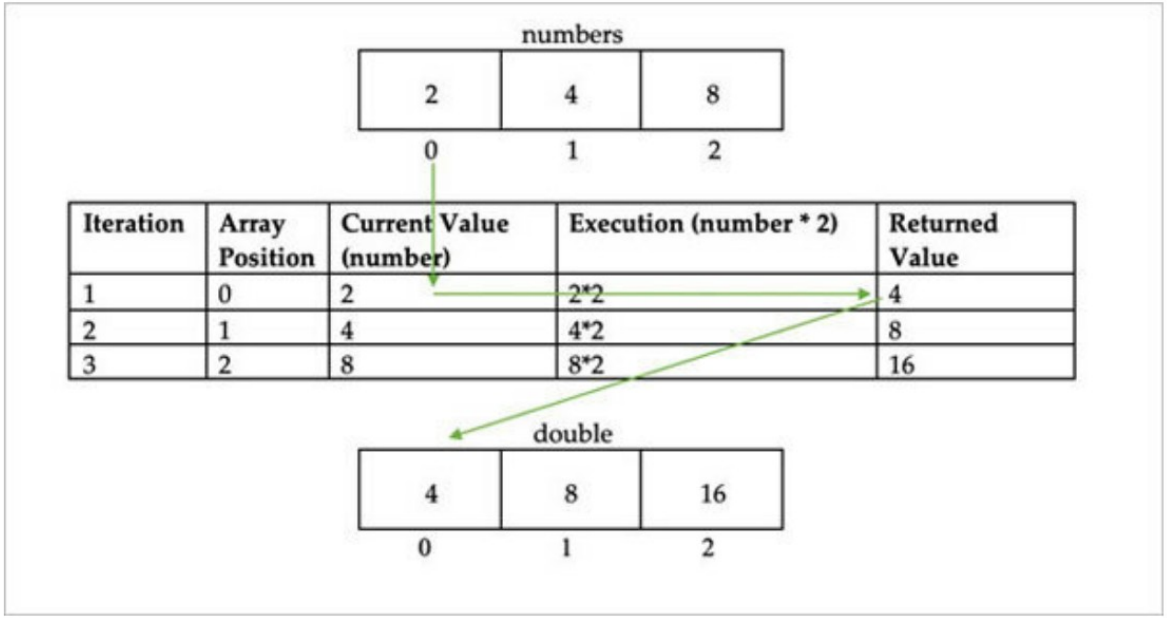

***Figure 6.37**: A visual reference to the map method*

Note: Use the map method only when you want to return an array. If you don't intend to return any array, then *do not use* the map method. Instead, you can use the `for...of` condition or `forEach` condtion. *6*

filter()

The literal job of the **`filter()`** method is to filter out elements from an array based on a condition. Imagine if you are provided an array of numbers. If I ask you to remove only those elements which are odd numbers, how will you do it?

The **`filter()`** method provides a short and a simple syntax to filter out elements in an array. The **`filter()`** method returns a new array of the filtered elements.

```
1. let numbers = [1, 20, 21, 46, 213, 452,
   615, 729, 8901];
2. let even = numbers.filter(number => number
   %2 === 0);


3.
4. console.log("Even numbers:     ",
   even);     // Even numbers:  [20, 46, 452]
```

***Code 6.12**: ES6 method to filter values*

```
1. let numbers = [1, 20, 21, 46, 213, 452,
   615, 729, 8901];
2. let even = numbers.filter(number => number
   %2 === 0);


3.
4. console.log("Even numbers:    ",
   even);    // Even numbers:  [20, 46, 452]
```

***Code 6.12**: ES6 method to filter values*

As shown in the preceding *code example 6.12*, once again, we make use of the arrow functions in the filter method of ES6. Like the map method, we call the filter method on the array numbers. The variable number refers to every single element inside the array numbers. This element is then checked against the condition shown after the fat arrow notation. If the condition is satisfied, that element is added to the new final array which will be returned. If the condition is not satisfied, that element is not added to the new final array.

reduce()

The **`reduce()`** method of ES6 reduces an array to a single value. The reduce method contains two parameters. The first parameter is an accumulator. In simple words, it will be the final value which will be returned as the result of the **`reduce()`** method. The second parameter is the current element of the array. Finally, after the fat arrow function, we execute some code on one or both the arguments. The resultant value is then stored in the accumulator (the first argument) for the next iteration. This process keeps on going until the method reaches the last element of the array.

Without much further ado, let's understand this with a simple example:

```
1. let numbers = [2, 4, 6, 8, 12];
2. let sum = numbers.reduce((total, number) =>
   total + number);


3.
4. console.log(`The sum is ${sum}`);       //
   The sum is 32
```

Code 6.13: *ES6 method to reduce an array to a single value*

```
1. let numbers = [2, 4, 6, 8, 12];
2. let sum = numbers.reduce((total, number) =>
   total + number);


3.
4. console.log(`The sum is ${sum}`);       //
   The sum is 32
```

***Code 6.13**: ES6 method to reduce an array to a single value*

In the preceding *code example 6.13*, we have called the reduce method on the array numbers. It will pass through every element of the array numbers.

We have named the two parameters as total and number respectively. The variable total is the accumulator, that is, it will store the collection of the calculated value. The second variable number refers to the current element of the array. Finally, after the fat arrow function, we add the two parameters and store the result in the accumulator, that is, the variable total in our case. Therefore, the reduce method passes through every element, adds the current value to the total and finally returns the accumulator total as the final value.

Array and object destructuring

The English dictionary states the meaning of the word destructuring as destroying the structure of something, or to dismantle something. In ES6, we use this concept of destructuring to break down values of an array into simple variables. We can think of it as unpacking a box of values.

Figure 6.38: *Destructuring is similar to unpacking items from a box*

Let's understand this 'unpacking' using examples:

Figure 6.39: *Destructuring an array*

As you can see in the preceding screenshot, we have initialised the array colors with four values of strings. Now, if you remember *Chapter 4* correctly, arrays are called as an organized collection. It means that an array has indexed values, that is, the emphasis is on the index or the position of the element.

In array destructuring, we mimic an array on the left hand side, and provide variables where we want to unpack and store certain values of the array. On the right hand side, we assign this mock array with the value of the original array from which we want to unpack the values. As you can see, the array destructuring matches the variables on the left hand side with the positions of the actual array colors, and assigns the variables on the left with their corresponding positions of the original array on the right.

In the case of object destructuring, since objects are not indexed collections, we cannot use the same method we used for the arrays. For objects, we need to provide the same variable names on the left that are actually present as keys in the object. Alternatively, we can provide these keys a pseudo-variable name using the colon symbol.

```
> let car = {
      name: 'Tesla',
      model: 'S',
      type: 'Electric'
  }

  let { type: carType, name, wheels } = car;
  console.log(carType, name, wheels);
  Electric Tesla undefined                                VM2530:8
```

Figure 6.40: *Destructuring an object*

As shown in the preceding screenshot, we can store the keys and the values of an object into individual variables using the same technique we have used for array destructuring. Instead of the mock array, we now use a mock object with the curly braces on the left side and the object on the right side of the assignment. The mock object contains the names of the keys of the object. As you can see in *figure 6.40*, we do not have a key called wheels in the original object car. Therefore, when we try to unpack the object using the variable wheels, it returns undefined. However, we can assign a new variable to the original key by using the colon, as we have used for the key name type.

ES6 Modules

In the early years of JavaScript, codes with thousands of lines of code were rare. Fast forward to 2020, and we have entire applications created using JavaScript. It's fair to say that across several pages of an application, there might be several blocks of code we might want to reuse. Sometimes, these blocks of code do not even belong to the flow of the code. For instance, we can have a function to add several numbers. Instead of writing the function add multiple times, we can define this function once and use it multiple times. This conversion of code into small reusable parts is called as modular coding and the small pieces of codes are called as modules.

Now, these modules might not even belong in the flow of the code. For example, a JavaScript file might be dedicated to the functionality of login. It's obvious that the add function does not belong in the middle of the logic of login. Besides, there might be many other JavaScript files that would need the assistance of this add function.

It, therefore, makes sense to store such a utility function in a separate JavaScript file of its own where it can be used by everyone. Now, the next problem here is how to make it available to every file in the system. This is where the ES6 imports and exports come into the picture.

To understand ES6 modules, imports and exports, we will create a sample **`index.html`** file:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.   <meta charset="UTF-8">
5.   <meta name="viewport" content="width=de-
   vice-width, initial-scale=1.0">
6.   <title>ES6 Modules</title>
7. </head>
8. <body>
9.   <p>The sum is <span id="result"></span></
   p>
10.  <script type="module" src="./scripts/in-
   dex.js"></script>
11. </body>
12. </html>
```

Code 6.14: *index.html for modules*

As you can see in the preceding *code example 6.14*, it's a simple **`index.html`** file with one major change – we are adding a type module to the import of **`index.js`** file in the HTML.

This is to make sure that the browser knows that this file will be using import and export in its code.

Now, let's create a folder called **`scripts`**, and inside that, let's create a file called **`utils.js`**. Inside **`utils.js`**, we will create out module function **`add`**.

```
1. export const add = (number1, number2) =>
   number1 + number2;
```

Code 6.15: *utils.js for the module*

As you can see in the preceding *code example 6.15*, we have now created the modular reusable function add using ES6 arrow functions. It takes two values, adds them and returns the sum. The only additional thing we have done here is that we used the keyword export. This keyword ensures that this function will be available to be used by all the files who would like to import it.

Now, finally, we will create the **`index.js`** file that we have mentioned in the **`index.html`**. It will look like the following code:

```
1. import {add} from './utils.js';


2.
3. let result = add(5, 6);
4. document.getElement-
   ById('result').innerHTML = result;
```

***Code 6.16**: index.js for the module*

In the **`index.js`** file, the first thing that we try to do is import the function **`add`** from the **`utils.js`** file. To do that, we use the keyword import, and then import the name of the function inside the curly braces. Finally, we complete the line by stating the path of the file from which we need to import this function. Why do we add the curly braces? We add the curly braces because every file might export several other functions. It's possible that we might want to use only a few of these functions. These functions can be imported by separating their names with commas.

If we would have added the keyword default after the keyword export in the *code example 6.15*, it would mean that the entire file is used for exporting only one function, which is the add function. In that case, we do not need the curly braces while importing.

The rest of the *code example 6.16* stores the value of the function execution of the function add, and assigns it to an empty span element in the HTML file.

I urge you to execute this file on your machine. This will help you to understand the code better.

```
1. export const add = (number1, number2) =>
   number1 + number2;
```

***Code 6.15**: utils.js for the module*

```
1. import {add} from './utils.js';
2.
3. let result = add(5, 6);
4. document.getElement-
   ById('result').innerHTML = result;
```

***Code 6.16**: index.js for the module*

Conclusion

In this rather long chapter, we learned about several features introduced by ES6. Singling out any feature would be a sin. However, to name certain ES6 features that you will be regularly using for creating next-gen applications include, but is not limited to, arrow functions, let and const, destructuring, and spread operators. To put it in a nutshell, functions like arrow functions and the new array methods like **`map`**, **`filter`**, and **`reduce`** not only reduced the number of lines of code, but it also simplifies and increases the speed of execution of JavaScript. Similarly, **`let`** and **`const`** help you to create scopes which can avoid major pitfalls while creating enterprise-level applications. The rest of the features help improve and facilitate faster execution of JavaScript. ES6 forms the basis of several top-notch front-end libraries, used for creating popular web applications like *Facebook*, *Twitter*, *Instagram*, and so on. In fact, libraries like *ReactJS* actively ask the developers to learn ES6 before learning React. React heavily uses several important features of ES6. In the next chapter, we will learn about classes. Classes is a JavaScript concept introduced by ES6 to simplify the creation of object templates.

Points to remember

- Hoisting in JavaScript is the process of moving declarations to the top. It is important to note that only the declarations, and not the initializations are moved to the top.
- ES6 introduced two new keywords for creating variables – **`let`** and **`const`**. Both **`let`** and **`const`** create a new type of scope called block-level scope. In a block-level scope, the variables' existence is limited solely to the block. These variables are not accessible outside the block. You cannot change the value set by **`const`**.

- The bind, call, and apply method for objects are used for binding the value of this for objects.
- The ternary operator is a simplified way for conditional if…else statement. Ternary operators are best used for cases when you need to return certain value.
- The advantage of rest parameters is when you are not certain about the number of parameters required in the function. Spread operators have the same syntax as that of the rest parameters. However, spread operators are used for concatenating or copying arrays or objects.
- Perhaps, one of the best features introduced by ES6 are the template literals. Replacing the traditional method of concatenating strings, template literals not only allow a simpler way of inserting variables in strings, but also allow multi-line strings.

- Arrow functions serve two significant purposes – simplifying the process of writing functions that return values, and removing the bindings of the `this` keyword inside functions.
- ES6 introduced three new methods of arrays – map, filter, and reduce. All the three methods parse through every element of the array, and should be used only when you need to return a new array. The map method allows you to manipulate every element of an array, the filter method allows you to filter elements of an array based on a condition, and the reduce method is to reduce all the elements of the array to a single value.
- Array and object destructuring is akin to unboxing values of an array or an object into separate variables.
- ES6 modules is a method of creating reusable functions that can be imported and exported across the application workspace.