HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Web Programming
JavaScript – node.js - react

Lecture 1-1:
JavaScript Update ES6+
19.10.

Stefan Noll
Manuel Fehrenbach
Winter Semester 22/23

ES6

Through its brief history, JavaScript has seen several versions and numerous updates. However, one of the most significant updates arrived in 2015. This update was called *ECMAScript2015* or, as it is more popularly known, ES6. This update turned up with some brilliant new features and methods. In fact, at the time of writing, several front-end libraries, which power applications like Facebook, Instagram, Twitter, etc, strongly recommend learning in-depth about ES6.

In this chapter, we will explore a few of the most important concepts introduced in ES6. If you complete this chapter by practicing the codes given, you should reach a stage where you can begin learning front-end libraries like React and Angular.

Structure

In this chapter, we will cover the following topics:

- Introduction to ES6
- JavaScript hoisting
- New ES6 syntax
- Arrow functions
- New array methods – map, filter, and reduce
- Array and Object Destructuring
- ES6 modules

Objective

After studying this chapter, you should be able to apply the latest and the most advanced methods in JavaScript to your application. You will learn about modules, which forms the basis of libraries like Angular and React and write optimised JavaScript code.

Introduction to ES6

Up until 2015, JavaScript was functioning primarily on version 3, released in 1999, and version 5, released in 2009. For a considerably long time, no new updates were added to the scripting language of web development.

Finally, in 2015, the sixth edition of JavaScript was released, and it was titled ECMAScript2015. For the sake of simplicity, this version became popular as ES6.

ES6 brought for a variety of new features and methods. A few of them are listed as follows:

- New keywords for creating variables – **`Let`** and **`Const`**.
- Object extensions like Bind, Call, and Apply.
- Arrow functions.
- JavaScript hoisting.
- Advanced operators like ternary operators and spread operators.
- Advanced array methods like map, filter, and reduce.
- Template literals.
- Module imports and exports.

JavaScript hoisting

As per the official documents of MDN Web Docs (it is pretty much like the Holy Bible for the web developers), JavaScript Hoisting was officially specified in the version ECMAScript2015, aka, ES6.

In layman terms, JavaScript hoisting is the default behavior of JavaScript to move declarations to the top. Let's understand this with an example.

```
1. number = 1;
2. console.log(number);
3. var number;
```

***Code 6.1**: An example of JavaScript Hoisting*

Traditionally, we declare the JavaScript variable first and then initialise it. However, in the preceding *code example 6.1*, we are initializing the variable before declaring it. Based on what we have learnt so far, what do you think will be the output of line number 2?

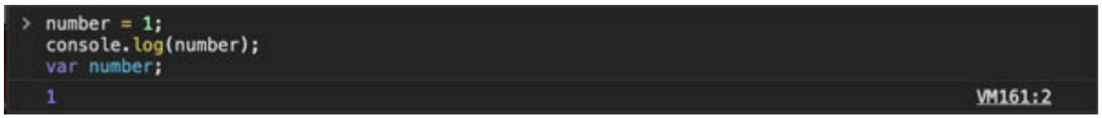
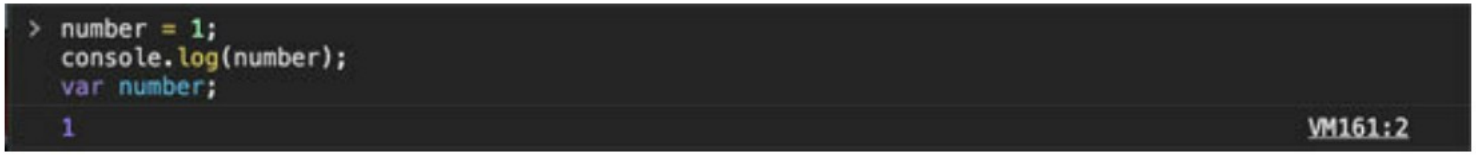


***Figure 6.1**: Output of the code example 6.1*



As you can see in the preceding screenshot, the output of the line number 2 in the *code example 6.1* is 1 and it did not throw an error. Why did this happen?

In technical terms, JavaScript puts the variable declarations and the function declarations into the memory during compile phase. In layman terms, no matter where you have declared your variable, it can be visualised as moving to the top of the code. Therefore, all the variable declarations in your code are executed first and then the JavaScript continues its normal execution line-by-line.

Therefore, in the *code example 6.1*, the declaration of the variable number on line number 3 happens before the first line is executed. Hence, when JavaScript executes line number 1, it already has an empty declared variable called number. JavaScript then initialises it with the value 1, as shown on the line number 1 of the *code example 6.1*. Since it has been both, declared and initialised, the execution of the line number 2 of the *code example 6.1* results in the value 1. Imagine this process as the task of filling water inside a container. To fill water inside a container (to add a value to a variable), it is obvious that we need to make sure that the container exists before we try to fill it with water (the variable exists before we add a value)

Figure 6.2: *JavaScript Hoisting as an example of container and water*

Let's understand the concept of hoisting with a few more examples:

```
> number1 = 25;
  console.log("number1:   ", number1);
  console.log("number2:   ", number2);
  console.log("number3:   ", number3);
  var number2 = 50;
  number1:    25                                          VM456:2
  number2:    undefined                                   VM456:3
✖ ▶Uncaught ReferenceError: number3 is not defined        VM456:4
      at <anonymous>:4:28
```

Figure 6.3: *Not defined vs undefined in JavaScript Hoisting*

So what on earth happened in the preceding screenshot? Let's understand it one line at a time.

Code Explanation:

Let's begin by understanding what happened to the variable number2:

Based on the prior explanation and the preceding screenshot, we understand that the variable number2 was declared and initialised after the statement to print it using **`console.log`**. However, since JavaScript moves all the declarations to the top, when the line number 3 (**`console.log("number2: ", number2);`**) is executed, JavaScript already has the declared variable number2.

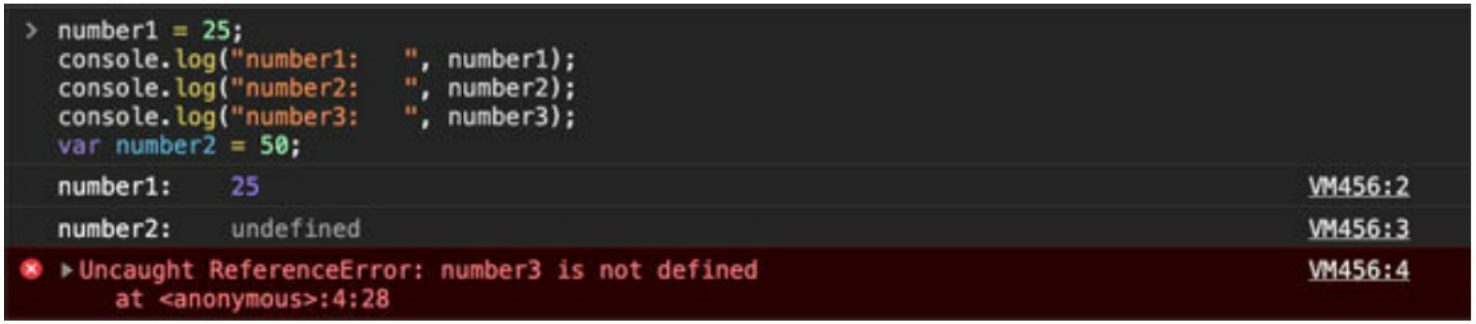
So, then why did it not print 50? Because hoisting only moves declarations to the top, it does not move the initialisations to the top. Imagine this as placing the container (declaration) before you fill water (the value 50) inside it. Therefore, when the line number 3 is executed, JavaScript holds an empty declaration of the variable number2 (or in the other sense, we have the container already placed but we have not yet filled the water inside it). Hence, it prints number2 as undefined.

```
> number1 = 25;
  console.log("number1:   ", number1);
  console.log("number2:   ", number2);
  console.log("number3:   ", number3);
  var number2 = 50;
  number1:    25                                          VM456:2
  number2:    undefined                                   VM456:3
✖ ▸Uncaught ReferenceError: number3 is not defined        VM456:4
      at <anonymous>:4:28
```

Now, let's understand what happened to the variable number1:

Unlike the *code example 6.1*, in *figure 6.3*, we initialise the variable number1 with the value 25, but never declare it, even after printing it on the console.

So, in that case, since the variable number1 is not declared and we are attempting to print it, it should throw an error, right? The answer is no. Call it intelligence, but since you are initialising the variable number1 with the value 25, JavaScript presumes that you want to declare it. Therefore, it automatically declares the variable number1, since it has been initialised. Consider the abstract example of the container and the water that we imagined earlier. If you tell someone that you are filling water inside a container, they will already presume that you do have a container.



So, why do we need to declare a variable at all? Why do we need to write **`var`** before we give it a value? The answer to these questions is that the keyword **`var`** helps us in defining the scope or the usability of the variable. In ES6, JavaScript introduces two new ways of declaring a variable, which will completely change the way we look at scopes. In the next few sections, we will understand the two new keywords (**`let`** and **`const`**) for declaring the variables, and that will answer the question here.

Finally, what happens to the variable number3?

If we look at our code, at no point in time do we initialise or declare the variable number3. It means that the variable number3 simply does not exist. In the line number 4, we are trying to print the value of something which does not exist. In every sense, it will throw an error saying that the variable does not exist or in JavaScript terms, it will say the variable number3 is not defined.

Like variables, JavaScript also moves function declarations to the top. Let's understand this with examples:

```
> add(10, 20);

  function add(number1, number2) {
      return number1 + number2;
  }
< 30
```

Figure 6.4: *JavaScript Hoisting with functions*

As shown in the preceding screenshot, even though we call the function add before it is declared and initialised, JavaScript still shows the output of the function execution. As we learnt earlier, JavaScript hoisting moves the function declaration to the top. In this case, we are also initialising the function by typing the code that needs to be executed when the function is called. Hence, we are initialising and declaring the code together.

Now, let's take a look at some erroneous coding with JavaScript hoisting and functions:

***Figure 6.5**: JavaScript Hoisting for functions with error*

In the preceding screenshot, we have declared a variable add. Even if this variable is placed after the line number 2, it is not initialised. Hence, it will have no value. At this point, it is not even a function. However, in line number 2, we consider the variable add to be a function and call it with the arguments 1 and 6. To state the obvious, it will throw an error stating that the variable add is not a function.

Now, let's take a look at one final example of functions and JavaScript hoisting:

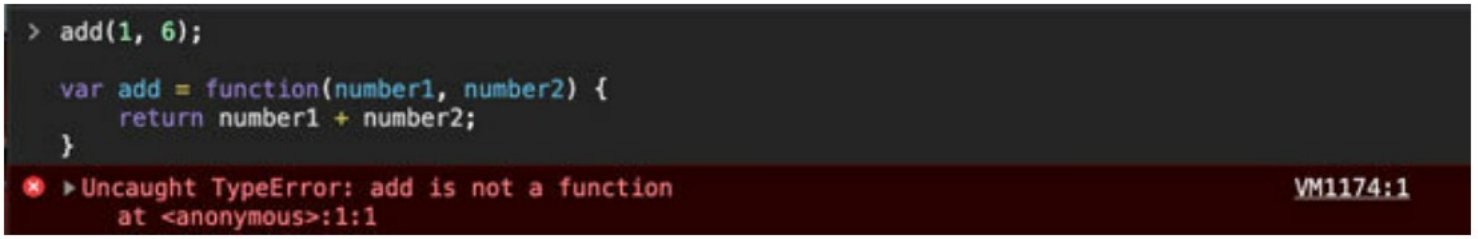


***Figure 6.6**: JavaScript Hoisting for anonymous functions*

In the preceding screenshot, we have used an anonymous function. We initialise the variable **add** with the anonymous function. However, as per JavaScript hoisting, only the variable declaration moves to the top. At the point of calling the anonymous function stored in the variable add with arguments 1 and 6, it is only an empty undefined variable. Hence, we get the error stating that add is not a function.

New ES6 Syntax

ES6 introduced several new syntaxes for JavaScript developers. These new syntaxes amplify the speed of code execution and optimise the JavaScript code to a great extent. Let's understand these new syntaxes one by one:

Let and const

As we have learnt earlier in *Chapter 3: Functions*, before the introduction of ES6, JavaScript had two types of scope: *global scope* and *local scope*. Variables declared with the JavaScript keyword **`var`** will have a local scope inside a function. Similarly, the variables declared with the JavaScript keyword **`var`** will have a global scope outside the functions (that is, the variables declared outside the function will be accessible by everyone).

ES6 introduces a new level of scope called block-level scope. You can create a block-level scope in JavaScript using the keywords **`let`** and **`const`**. The word block in the term block-level scope means a block of code. This block could be a block of code inside a pair of curly braces, a block of code defined inside a function, or a block of code in the entire script. The most important thing to understand about these blocks is that each block is a separate entity and has its own private space. Let's understand what block-level scope is, and how to create it.

let

In JavaScript, if you declare a variable with the keyword **`var`** inside curly braces and access the variable outside, you will be able to access the variable.

Figure 6.7: *Scope by keyword var*

As shown in the preceding screenshot, we have declared a variable number and initialised it with a value 20 inside curly braces. When we try to access the variable number outside the curly braces and AFTER the variable definition, we are able to access the value, as shown in the output.

In an ideal scenario, the variables declared inside curly braces should NOT be accessible outside the braces. Let's consider a practical example for this scenario:

```
> var number = 25;
  console.log("1. Before curly braces:   ", number);
  {
      console.log("2. Inside curly braces before redeclaration:  ", number);
      var number = 50;
      console.log("3. Inside curly braces after redeclaration:   ", number);
  }
  console.log("4. After curly braces:    ", number);
  1. Before curly braces:    25                                    VM765:2
  2. Inside curly braces before redeclaration:   25                VM765:4
  3. Inside curly braces after redeclaration:    50                VM765:6
  4. After curly braces:     50                                    VM765:8
```

Figure 6.8: *Why var can be complicated!*

In the preceding screenshot, we have defined a variable number with the value 25 on line number 1. After that, we have opened curly braces and redefined the variable number with the value 50.

As we learnt earlier, the curly braces resemble a block. The values inside this block should ideally remain accessible only inside the block. Similarly, since the code outside the curly braces forms a separate block, we would imagine the values declared outside the curly braces (block) to retain their respective values outside. However, as we can see in *figure 6.8*, if a user has redefined a variable with another value inside the curly braces, the old value gets completely replaced everywhere after the redeclaration.

Now, imagine a script of approximately 1000 lines of code. It's practically impossible to remember the names of all the variables used in a script. If a developer accidentally uses the same variable name to define a new value, there is every chance that the code might crash.

```
> var name = 'John Doe';
  {
      name = 'Mercedes Benz';
  }
  console.log("Name:    ", name);
  Name:    Mercedes Benz                                    VM1421:5
```

Figure 6.9: *Using same variable name with var changes the value*

Imagine the user has defined the variable title for storing the name of the user. However, somewhere in the middle of the script, the developer also uses the same variable name title for storing the name of the user's car. Now, even though the developer must have planned to initially reserve the variable name title for storing the name of the user, it will show the name of the user's car wherever it is used after the redefinitior

To solve such erroneous coding, ES6 introduced two new JavaScript keywords to create variables – **`let`** and **`const`**. The keyword let has the following four features:

- **Variables declared inside a block cannot be accessed outside the block**

In analogy with *figure 6.7* that we saw earlier, let's understand the same example by using the keyword let instead:

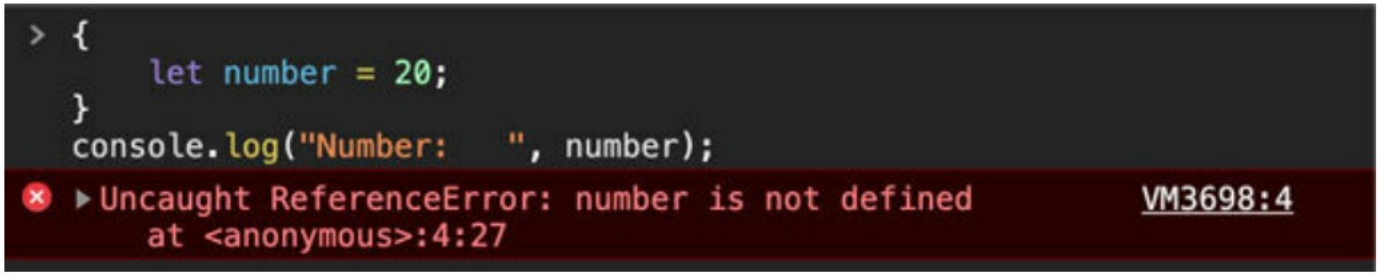


***Figure 6.10**: Variables created with let keyword is not accessed outside*

As shown in the preceding screenshot, the variable `number` created with the keyword `let` inside the curly braces (a block of code) cannot be accessed outside the curly braces.

- **Variables declared outside the block cannot be accessed inside the block**

Similarly, in analogy with *figure 6.8*, we will use the same example with the keyword let.

As shown in the following screenshot, you cannot access a variable inside the curly braces (block) if it is declared outside the block and a new variable with the same name is not redefined inside the block. Since the code outside and the code inside the curly braces form different blocks, value inside one block cannot be accessed inside another block.

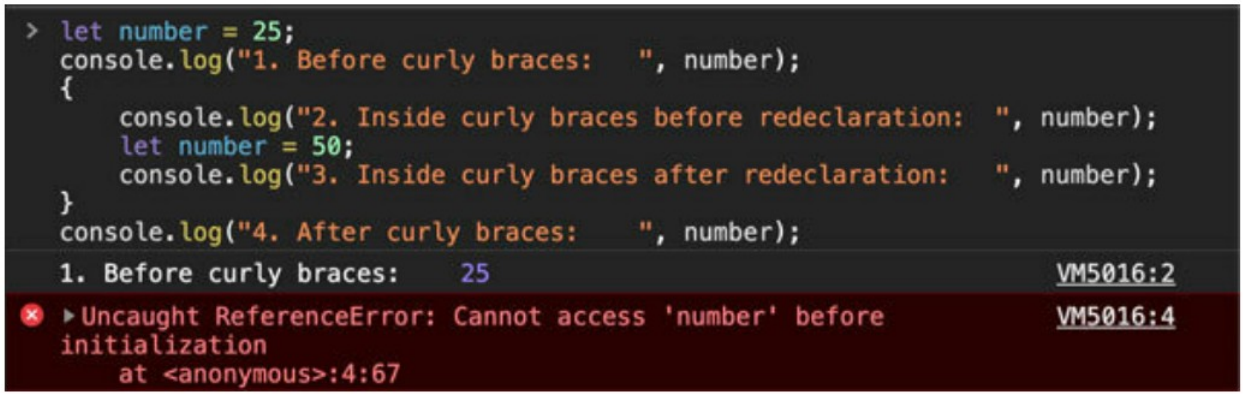


Figure 6.11: *Variables declared with let outside cannot be accessed inside*

- **Variables retain their values in their block**

  In conjunction with the previous bullet point, every block will retain their own value. As you can see in the following screenshot, the variable number declared outside the curly braces will retain its value after the curly braces end. Similarly, the value inside the curly braces will not impact the value outside.

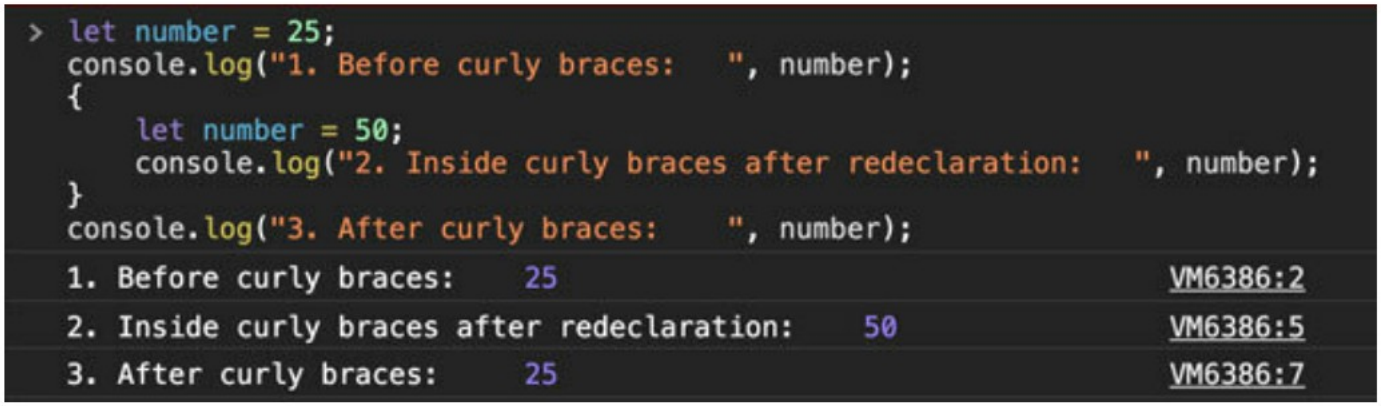


Figure 6.12: *Each block has its own value*

- **Variables with the same name cannot be redeclared inside a block**

Finally, one of the most important features of the keyword let is that it does not let you create variables with the same name inside a block. However, you can create variables with the same name in another block. As shown in the following screenshot, you are allowed to redeclare the variable number inside another block, as shown on the line number 4 in *figure 6.13*. However, if you try to redeclare the variable number after the curly braces, you are not allowed to do that. It's not allowed because the code outside the curly braces forms a different block, and the variable number is already defined on the line number 1 in that block.

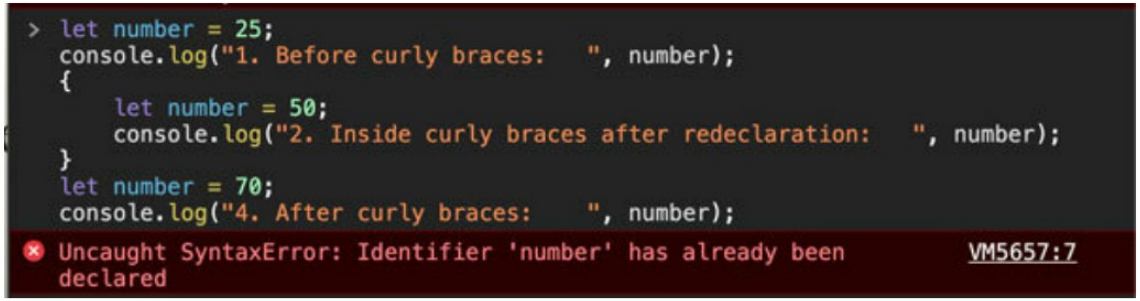


Figure 6.13: *Variables cannot be redeclared inside the same block*

Thus, the keyword let allows you to create a block-level scope for variables. This would solve the problem we encountered in *figure 6.9*. Why don't you test the script we saw in *figure 6.9* by using the keyword let instead of **`var`**, on your own?

```
> var name = 'John Doe';
  {
      name = 'Mercedes Benz';
  }
  console.log("Name:    ", name);
  Name:    Mercedes Benz                                    VM1421:5
```

Figure 6.9: *Using same variable name with var changes the value*

const

The variables declared with the keyword **`const`** work in similar fashion to the variables created with the keyword **`let`**. They also follow the laws of block-level scope. However, a major difference between the two is that you cannot reassign a different value to the variables declared with const. In simple words, you create a constant value that cannot be changed.

```
> const number = 25;
  number = 50;
✕ ▸Uncaught TypeError: Assignment to constant variable.        VM9519:2
      at <anonymous>:2:8
```

Figure 6.14: *Variables cannot have new value using const*

The general convention for writing constants is by writing the variable name in uppercase and separating multiple words using underscores.

A few things that we need to know about **`const`** are as follows:

- **You cannot assign a new value to variables declared with const**

  As shown in *figure 6.14*, you cannot assign a new value to the variables defined with **`const`**.

- **You cannot reassign an already declared variable with const**

  Since **`const`** follows the same principle as let, you cannot redefine an already declared variable with **`const`**.

```
> var number = 25;
  const number = 50;
Uncaught SyntaxError: Identifier 'number' has already been declared    VM10957:2
```

Figure 6.15: *Variables cannot be redeclared with const*

A few things that we need to know about **`const`** are as follows:

- **It is *compulsory* to assign a value when using const**

  As shown in the following screenshot, you always need to initialise a variable declared with const. If you don't initialise a variable when using **`const`**, it will result in an error:

Figure 6.16: *You need to initialise a variable with const*

A few things that we need to know about **const** are as follows:

- **You can change the properties of objects declared using const...**

  As shown in the following screenshot, you can change the properties of an object declared with the variable const:

```
> const person = {
      name: 'John Doe',
      age: 45
  }
  console.log("Person before changing age:  ", person);
  person.age = 50;
  console.log("Person after changing age:  ", person);
  Person before changing age:   ▶{name: "John Doe", age: 45}      VM1918:5
  Person after changing age:    ▶{name: "John Doe", age: 50}      VM1918:7
```

Figure 6.17: *Properties of objects declared with const can be changed*

- **...but you cannot assign a new object to the same variable**

  As shown in the following screenshot, you cannot assign a new object to the variable declared with const:

```
> const person = {
      name: 'John Doe',
      age: 45
  }
  console.log("Person:   ", person);
  person = { name: 'Jane Dias', age: 26};
  Person:    ▶{name: "John Doe", age: 45}      VM3197:5
  ⊗ ▶Uncaught TypeError: Assignment to constant variable.      VM3197:6
      at <anonymous>:6:8
```

Figure 6.18: *You cannot assign a new object to the same variable using const*

Tip: To overcome the flaw where the properties of an object declared with `const` can be changed, you can use `Object.freeze()`.

```
> const person = { name: 'John Doe', age: 45 };
  Object.freeze(person);
  person.age = 50;
  console.log(person);

  ▶ {name: "John Doe", age: 45}
```

Figure 6.19: *Object.freeeze*

As shown in the preceding screenshot, if an object has been passed as an argument inside `Object.freeze`, it will not change its value. However, remember `Object.freeze()` does not work on objects created inside an object.

Bind, call, and apply

ES6 introduces three new methods for objects – bind, call, and apply. Let's directly delve into understanding these three methods with the help of examples:

```
> let person = {
      firstName: 'John',
      lastName: 'Doe',
      getFullName: function() {
          console.log(this.firstName + this.lastName);
      }
  }

  var fullName = person.getFullName;
  fullName();
  NaN                                                    VM509:5
```

Figure 6.20: *Object method returns NaN when called*

In the preceding screenshot, execution of the script results in the value NaN. Any guesses why?

Let's revisit the concept of the **`this`** keyword. We have learnt that the **`this`** keyword corresponds to the owner of the function. In the preceding case, when the object is defined, the this keyword inside the method **`getFullName`** refers to the **`person`** object. However, on the line number 9, we assign the method to the variable **`fullName`**. Since the variable **`fullName`** belongs to the global object, when the variable **`fullName`** is called, the value of this shifts to the global object. And since the global object does not have variables **`firstName`** and **`lastName`**, we get the value NaN.

To resolve this, JavaScript provides three new methods for the global object **`Object`**. Let's understand every method one-by-one.

```
> let person = {
      firstName: 'John',
      lastName: 'Doe',
      getFullName: function() {
          console.log(this.firstName + this.lastName);
      }
  }

  var fullName = person.getFullName;
  fullName();
  NaN                                                    VM509:5
```

bind

The bind method creates a new copy of the function. It accepts a value as an argument. The bind function sets the value of the **`this`** keyword to the value provided in the argument. Building on the example we saw earlier in *figure 6.20*, we pass the value of the object person to the bind function in the line number 9, as shown in the following screenshot:

***Figure 6.21**: The bind method*

In the preceding screenshot, we pass the object person as an argument to the bind method in the invocation of the **`getFullName`** method. The bind method creates a new copy of the original method **`getFullName`** and sets the **`this`** keyword to the person object.

call

The call method does the same thing as the bind method. However, there are three notable differences between the two methods:

- The call method does not create a new copy of the function.
- The call method immediately executes the function.
- The bind method accepts arguments.

Let's understand this with an example.

```
> let person = {
      getFullName: function() {
          console.log(this.firstName + " " + this.lastName);
      }
  }

  let person1 = { firstName: 'John', lastName: 'Doe' };
  let person2 = { firstName: 'Jane', lastName: 'Dias' };

  person.getFullName.call(person1);
  person.getFullName.call(person2);
  John Doe                                                    VM1331:3
  Jane Dias                                                   VM1331:3
```

Figure 6.22: *The call method*

If you compare *figures 6.21* and *6.22*, you will notice that in *figure 6.21*, you had to invoke the function separately after applying the bind method. In *figure 6.22*, you can notice that you don't need to separately invoke the function. Instead, the call method, by default, calls the function. Another important thing to notice is that the call method does not create a new copy of the function. For these two reasons, the call method is used for applying methods on different objects.

In the following screenshot, you can observe another feature of the call method. In the arguments that you pass to the call method, after you pass the object required for the **`this`** keyword, you can also pass multiple comma-separated arguments to be used by the method.

```
> let person = {
      getFullName: function(age, nationality) {
          console.log(this.firstName + " " + this.lastName + " is a " + age +
  " year old " + nationality + ".");
      }
  }

  let person1 = { firstName: 'John', lastName: 'Doe' };
  let person2 = { firstName: 'Jane', lastName: 'Dias' };

  person.getFullName.call(person1, 27, "English");
  person.getFullName.call(person2, 26, "French");
  John Doe is a 27 year old English.                    VM2239:3
  Jane Dias is a 26 year old French.                    VM2239:3
```

Figure 6.23: *The call method with arguments*

```
> let person = {
      firstName: 'John',
      lastName: 'Doe',
      getFullName: function() {
          console.log(this.firstName + this.lastName);
      }
  }

  var fullName = person.getFullName.bind(person);
  fullName();
  JohnDoe                                               VM2655:5
```

Figure 6.21: *The bind method*

```
> let person = {
      getFullName: function() {
          console.log(this.firstName + " " + this.lastName);
      }
  }

  let person1 = { firstName: 'John', lastName: 'Doe' };
  let person2 = { firstName: 'Jane', lastName: 'Dias' };

  person.getFullName.call(person1);
  person.getFullName.call(person2);
  John Doe                                              VM1331:3
  Jane Dias                                             VM1331:3
```

Figure 6.22: *The call method*

apply

The apply method is similar to the call method with just one difference. In the apply method, you need to pass the extra arguments as an array instead of comma-separated values.

```
> let person = {
      getFullName: function(age, nationality) {
          console.log(this.firstName + " " + this.lastName + " is a " + age +
  " year old " + nationality + ".");
      }
  }

  let person1 = { firstName: 'John', lastName: 'Doe' };
  let person2 = { firstName: 'Jane', lastName: 'Dias' };

  person.getFullName.apply(person1, [27, "English"]);
  person.getFullName.apply(person2, [26, "French"]);
  John Doe is a 27 year old English.                    VM2462:3
  Jane Dias is a 26 year old French.                    VM2462:3
```

***Figure 6.24**: The apply method*

If you compare *figures 6.23* and *6.24*, you will notice that the only difference is that the arguments are supplied as an array in the latter. The benefit of supplying the arguments as an array is that you will always know that there will be only two maximum arguments to be passed inside the apply method. Additionally, you can simply outsource the array creation to a variable holding an array.

Advanced operators

Ternary operator

The ternary operators simplify the way we write an if…else conditional in JavaScript. Let's try to understand the ternary operator with a simple if…else example:

```
1. let result;
2. if(number%2 === 0) {
3.   result = "The number is even.";
4. } else {
5.   result = "The number is odd.";
6. }
```

***Code 6.2:** An if…else conditional in JavaScript*

In the *code example 6.2*, we have introduced a simple JavaScript if…else conditional to find out whether a number is even or odd. However, as you can see in the *code example 6.2*, it takes five lines of code. Even if we cut down on white spaces, we will still be typing a lot of characters.

To simplify the if…else conditional, ES6 introduces the ternary operator. The syntax of a ternary operator is as follows:

```
1. condition ? ifTrue : ifFalse
```

Code 6.3: *Syntax of ternary operators*

As shown in the *code example 6.3*, the syntax of the ternary operator requires two symbols – a question mark '?' and a colon ':'. In a ternary operator, you need to add the condition to test before the question mark. If the result of the condition is true, then the statement or the expression after the question mark (denoted by **`ifTrue`** in the *code example 6.3*) will be executed. If the evaluation of the condition is false, the statement or the expression after the colon symbol will be executed (denoted by **`ifFalse`** in the *code example 6.3*). Let's understand this syntax with the code shown in the *code example 6.2*:

```
1. let result = number%2 === 0 ? "The number
   is even" : "The number is odd";
```

Code 6.4: *Example of ternary operator*

As shown in the preceding *code example 6.4*, thanks to the ternary operator, we converted the seven lines of code in the *code example 6.2* to a single line of code in the *code example 6.4*.

Ternary operators work best when you need to return a single line of expression or a simple execution.

Let's look at one more practical example of ternary operators:

```
> function greeting(person) {
      return person ? person.name : "stranger";
  }

  console.log("Hello " + greeting({name: 'John Doe'}));
  console.log("Hello " + greeting(null));
  Hello John Doe                                          VM658:5
  Hello stranger                                          VM658:6
```

Figure 6.25: *Example of ternary operator*

As you can see in the preceding screenshot, the ternary operator can handle null values. If the condition evaluates to null or if the expression received in the condition is null, it automatically executes the else part (the part after the colon). In *figure 6.25*, since the ternary operator received null in the second invocation of the function, it returns stranger.

Rest Parameters

Before we set out to explore the rest parameters, let's reiterate over a few important terms of JavaScript functions. In JavaScript functions, the variables that you create in the parentheses during the function declaration are called as parameters and the variables or the values that you pass inside the parentheses when you call (invoke) the function are called as *arguments*. In the following screenshot, we can see the difference between parameters and arguments in a visual format:

Figure 6.26: *Parameters vs arguments*

Now that we know the difference between parameters and arguments, let's understand the concept of rest parameters introduced in ES6. To understand rest parameters, let's look at the example of the function **`add`**, shown in *figure 6.26*.

In *figure 6.26*, we have a simple function which adds two numbers.

But, what if we needed a function to add three numbers?

We will need to create a brand new function which has three parameters.

But, what if we needed a function to add four numbers?

We will need one more new function which has four parameters.

What if we need a function where the number of parameters can vary? For instance, I need a function which can take any number of arguments, and it will still compute the results. I could write add(2, 3) or I could write add(4, 5, 6, 102, 12, 23), and I will get the results in both the cases.

This can be achieved using the rest parameters. The rest parameters have a prefix of three dots. This notation allows you to introduce indefinite number of arguments in a function. These arguments will be represented as an array. As usual, let's try to understand this with an example.

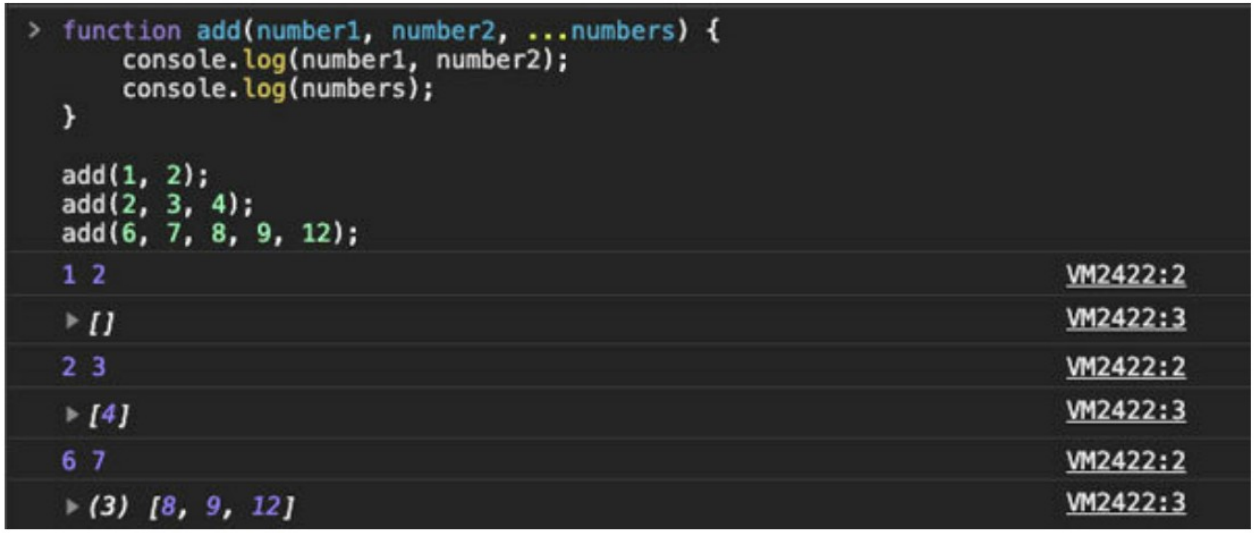


Figure 6.27: *Rest parameters*

As you can see in the preceding screenshot, we have created a function which takes two primary arguments number1 and number2. In the function declaration shown in the first line of *figure 6.27*, after typing the parameters number1 and number2, we have introduced a rest parameter denoted by a prefix of three dots (...) and a variable by the name numbers. This rest parameter numbers can store unlimited parameters in the form of an array. With the help of the rest parameters, now we don't need to create multiple new functions for different number of parameters. In fact, the user can now supply any number of arguments and our function will be able to handle all of them.

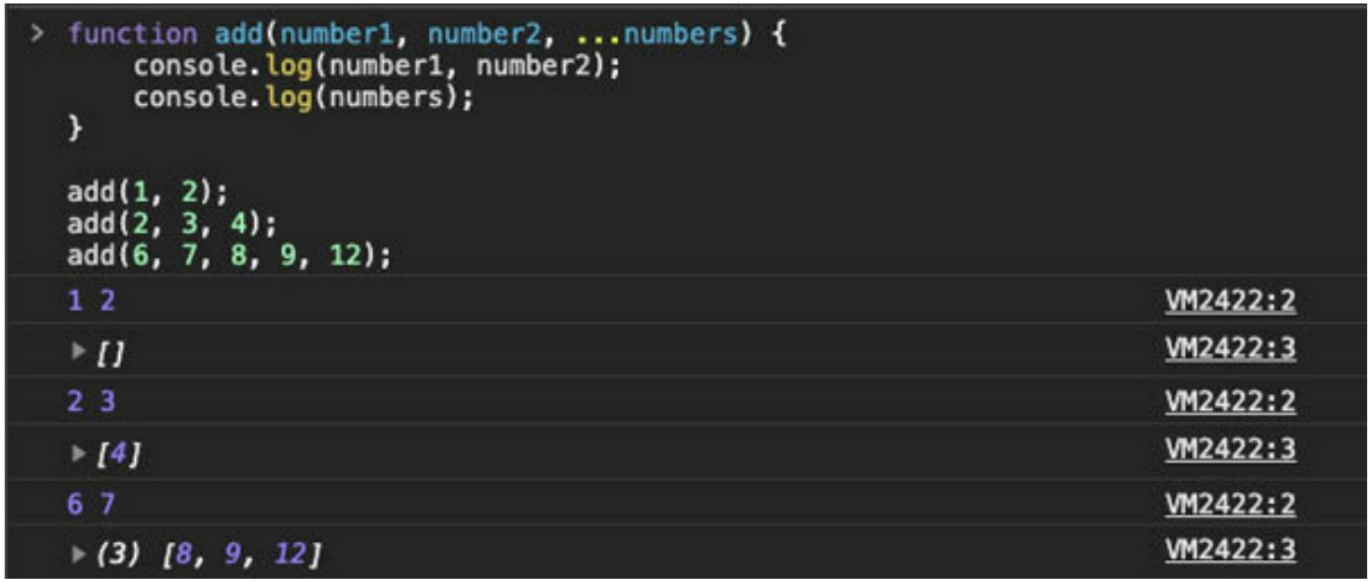


Figure 6.27: *Rest parameters*

In simple words, to use our function add, a user has to provide a minimum of two arguments.

As you can see in *figure 6.27*, if the user supplies only two arguments, as in, **`add(1, 2)`**, the rest parameter numbers will show an empty array. If the user supplies three arguments, as in, **`add(2, 3, 4)`**, we will have the array numbers with one value which is the third argument 4. Finally, if the user supplies five arguments, as in, **`add(6, 7, 8, 9, 12)`**, we will have the array numbers of three values 8, 9, and 12.

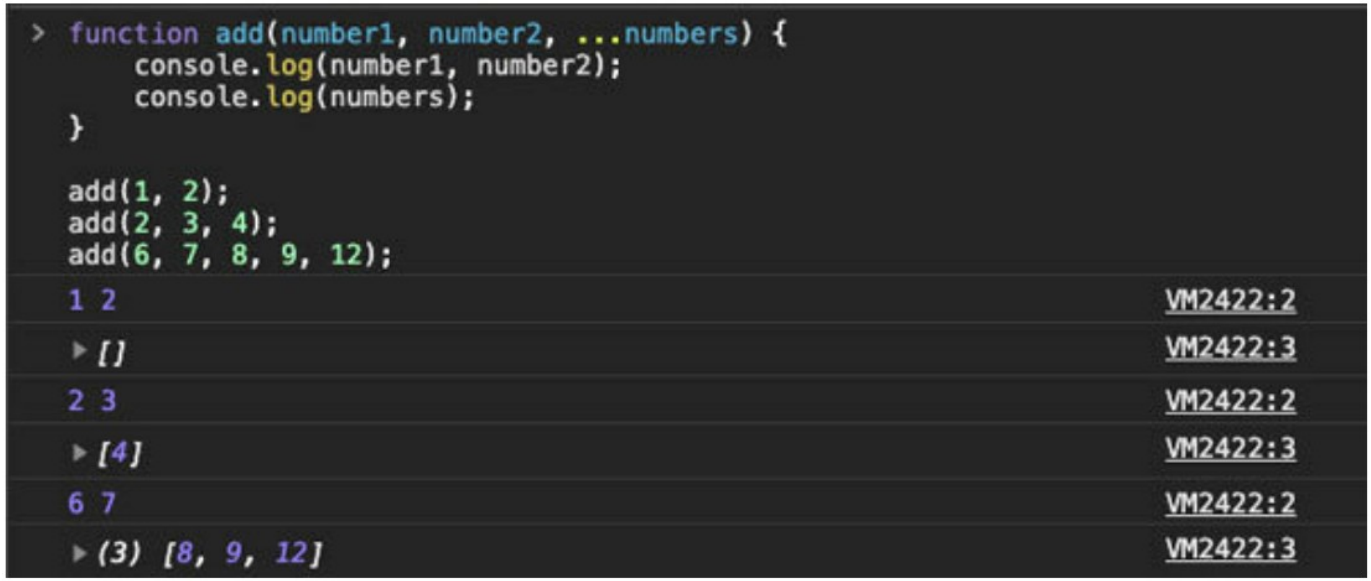


Figure 6.27: *Rest parameters*