

Programming JavaScript

Santosh Kalwar, 31.01.2022

JavaScript Topics

- Semi-Colons (optional)
- Implicit return
- Practice
- Strings and Array (recap + advanced)
- Practice
- Array reduce
- Practice
- Optional Chaining
- Practice
- Supermini projects

Semi-Colons

If you've already seen some JavaScript code somewhere on the Internet, you may have noticed that most lines end with a semi-colon (;).

And you may even have wondered why we have kept it optional not using those semi-colons.

In this chapter, we're going to answer the following questions:

- Why do semi-colons exist in JavaScript?
- Whether or not I recommend using semi-colons
- When using semi-colons, when should you put them in and when you shouldn't

Why does JavaScript have semi-colons?

Not all programming languages have semi-colons but JavaScript does, and it's for a very good reason.

When you develop a website, you will be writing a lot of JavaScript code. In order to make the website smaller, we often apply a concept called **minification** whereby you use a tool to minify your code in order to save characters. Saving characters automatically saves KBs. For example:

```
/**
 * @param {string[]} items
 * @param {string} item
 */
const addItem = (items, item) => {
  items.push(item)
  return items
}

/**
 * @param {string[]} items
 */
const exportLowerCasedCSV = items => {
  return items.map(function(item) {
    return item.toLowerCase()
  }).join(", ")
}
```

Why does JavaScript have semi-colons?

will be minified to the following:

```
function addItem(items, item) {items.push(item);return items;}function  
exportLowerCasedCSV(items) {return items.map(function(item) {return  
item.toLowerCase();}).join(", ");}
```

The code above is barely readable however it is 122 characters shorter (so that is 122Bytes smaller). This is a small example, but for big JavaScript files, you can see as much as a 40% reduction in file size.

Minification removes all the comments in the code and then removes all the blank spaces.

If you look closer, you will see that after every JavaScript "line", we now have a semi-colon. So the goal of semi-colons is to tell JavaScript where a line ends. That's because when we minify the code, there are some edge cases that end up breaking your code. Semi-colons are here to solve that. They signify the end of a statement.

Why does JavaScript have semi-colons?

Note: you should **never** write minified code yourself, this is a process that is automated with tools such as `webpack` or `parcel`, which must be covered in React course / We will look into examples of how to use `parcel` later in the course.

Always write code on multiple lines, add comments whenever you need it. Tools will then strip them out for when you publish your application to the world.

Chapter Recap

- JavaScript has semi-colons because it's common to minify the code before publishing the website.
- Automated tools will minify your code. You should never minify it yourself or write minified code.
- Minification works by removing the comments and removing all the blank spaces (thus putting all your code on 1 line).

Should you use semi-colons?

The answer to that question is not a **yes** or a **no**, the answer is **it depends**.

It is possible that you write code while never using semi-colons because then tools such as `webpack` or `parcel` will insert those semi-colons for you during the minification process.

However, a lot of developers are used to using semi-colons (including myself!).

Another thing you should consider is your working environment. If you work at a company, they will have some set of guidelines on how to write JavaScript. Some companies use semi-colons, others don't. From experience, the majority of developers use semi-colons.

So my personal recommendation is to use semi-colons. But understand that it is not a big deal. So don't spend too much time on this debate because it's a debate and it will always be.

If you already know another programming language and you're already used to semi-colons, then just use them! If you don't like to use them, then don't use them! Another reason why it's not a big deal is that you can use tools such as **prettier** that will automatically format your code. It will also automatically insert semi-colons in your source code (before the minification process).

Prettier

Prettier is an opinionated code formatter, which means that the creator of prettier has made some decisions on your behalf to format your code (which you can then override if you don't like them). A lot of developers are very happy with Prettier which is why it has recently become quite popular. We recommend that you set up prettier on your computer. Here are the instructions for setting it up for VSCode (including instructions for setting up VSCode):

- 1.If you don't already have VSCode installed, install it from [**code.visualstudio.com**](https://code.visualstudio.com).
- 2.After you open VSCode, click on the **Extensions** tab and then search for **Prettier** and click on the first result "Prettier - Code formatter".
- 3.Click on the **Install** button.
- 4.On the same page, scroll down a little bit and follow the remaining instructions under **Default Formatter** to ensure that Prettier is enabled in JavaScript files.
- 5.Navigate to the settings page (**cmd** , on mac or **ctrl** , on other platforms) and then search for **Format on save** and tick the checkbox.

VSCode will now automatically format your code using Prettier when you save it.

Prettier inserts semi-colons by default. But, like many of its options, it can be disabled.

I recommend that you keep all the default prettier settings unless a particular style is bothering you.

Chapter recap

- You can decide to use semi-colons or not based on your own preference.
- If/when you work at a company, you will be instructed to use semi-colons or not based on the team's preferences.
- Tools (such as **prettier**) can automatically insert semi-colons in your source code.

ESLint (Optional)

On the topic of code format, there's another related concept which is called **code style**.

Code style is the set of rules and guidelines that a certain team/person/company uses while developing a project.

You don't necessarily need to have a code style, however, having a code style can make your code less prone to bugs, prevent some bugs ahead of time, all while making your code more homogeneous.


Having homogeneous code in a big team is important to improve the long-term maintenance of the project.

The most popular tool for enforcing/recommending certain code styles in JavaScript is called **ESLint**.

Setup ESLint for VSCode (Optional)

1. Open VSCode, click on the **Extensions** tab and then search for **ESLint** and click on the first result "ESLint".
2. Click on the **Install** button.
3. At the root of your project, create a file called `.eslintrc.json` (don't forget the leading dot character)
4. Write the below inside this file and save it

```
{  
  "extends": "eslint:recommended",  
  "parserOptions": {  
    "ecmaVersion": 2021  
  }  
}
```

This uses the recommended ESLint rules which you can find on the [rules](#) page. All the ones marked with a check  are included in the `eslint:recommended` configuration.

Chapter recap

- Code style is the set of rules and guidelines that a certain team/person/company uses while developing a project.
- The most popular tool for enforcing/recommending certain code styles in JavaScript is called **ESLint**.
- <https://eslint.org/docs/rules/>

Automatic Semi-colon insertion

ASI (Automatic Semi-colon Insertion) is a concept in JavaScript that we'll explain in this lesson.

Some specific statements in JavaScript **need** to end with a semi-colon. So, if the programmer left them out, the JavaScript engine will **automatically place a semi-colon**. This is called Automatic Semi-colon Insert and only applies in very specific cases.

Here are the most common statements where ASI applies:

- let/const variable declarations.
- import/export statements.
- return.

You can view the full list on MDN:



ASI on MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#automatic_semicolon_insertion

Automatic Semi-colon insertion

You can write JavaScript for years and years without knowing about the concept of ASI. However, developers mostly encounter it when they start working with React and write code that looks like the following:

```
function App() {  
  return  
    <div>  
      Hello  
    </div>  
}
```

Note that the syntax above is neither JavaScript nor HTML, it's called JSX and is outside of the scope of this JS course. If you've never worked with React, don't worry about the example.

Automatic Semi-colon insertion

The problem with the code above is that JavaScript will automatically place a semi-colon after the `return` keyword, so this function effectively returns `undefined`. Here's how it'll look like after the JavaScript engine performs ASI:

```
function App() {  
  return ; // (similar to return undefined)  
  <div>  
    Hello  
  </div>  
}
```

This is where most developers discover the concept of ASI.

The solution would be to have the return on one line: `return <div>` and then the rest can go on the different lines. ASI, would not break it because the semi-colon would go at the end of the expression.

Alternatively, you can solve it by opening a parenthesis after the `return` keyword, which tells the JavaScript engine that it should expect an expression, thus, it'll place the semi-colon after the closing matching parentheses:

```
function App() {  
  return (  
    <div>  
      Hello  
    </div>  
  )  
}
```

Chapter recap

- Some specific statements in JavaScript **need** to end with a semi-colon. So, if the programmer left them out, the JavaScript engine will **automatically place a semi-colon**. This is called Automatic Semi-colon Insert and only applies in very specific cases.
- The most common example of ASI is with the `return` keyword (mostly while writing React code with JSX).

JS in just 5 minutes

Let us start today's day with a 5 minutes video

https://www.youtube.com/watch?v=c-l5S_zTwAc&ab_channel=AaronJack

Implicit return

When you forget to write `return` in a function in JavaScript, you get an implicit `return undefined`. The word implicit here means that it is inferred but not specifically expressed. Meaning that there was **no** `return undefined` but we end up getting `undefined` as a result.

For example:

```
const sum = (a, b) => {  
  a + b;  
}
```

```
sum(1, 3); // undefined
```

Implicit return

As you can see, `sum(1, 3)` returns `undefined` because we did not have a `return` keyword inside the function body.

However, with arrow functions, you can implicitly return the result of the function on some very specific conditions. Let's start with an example and then we'll explain how we got there:

```
// this works 👍 (implicit return)
const sum = (a, b) => a + b;
```

```
sum(1, 3); // 4
```

So, how come this example works and the previous one doesn't? Both of them do not have the `return` keyword. One of them works and the other one returns `undefined`. This is because, for the implicit return to work, you should have **all** the following conditions:

1. Your function should be an arrow function.
2. The function body should be only **one line**. This means you have to remove the curly braces.
3. You have to remove the `return` keyword because the function body is one line. When all these conditions are met, the arrow function will implicitly return the result of its function body (which is only one line).

Implicit return

Let's take another example:

```
const isLegal = (age) => {  
  return age >= 18;  
}
```

Now let's make use of implicit return by:

- 1.removing the curly braces
- 2.removing the `return` keyword

```
const isLegal = (age) => age >= 18;
```

Here's how you can read it:

`isLegal` is a function that takes the `age` and returns the result of the expression `age >= 18`.

Only use implicit return when the function body is one line and short. Never sacrifice code readability and clarity in order to use a certain feature.

Implicit return

When you have only one parameter, you are able to drop the parentheses around the parameter. The code above becomes:

```
const isLegal = age => age >= 18;
```

This is the shortest possible way (yet clear) you can write this function.

Practice / Classroom coding

Go to your itsLearning site, click on Programming JavaScript —> Select Practice folder —> select **Practice_lessons3101**

implicit01.js: Re-write the function `triple` without using the keyword `return`.

implicit02.js: Create a function `multiply` that returns the product of both of its parameters without using the keyword `return`.

implicit03.js: Complete the function `getPositiveTemperatures` such that it returns an array containing the positive temperatures (the temperatures that are above 0). Use an arrow function (implicit return is optional).

implicit04.js: Complete the function `getFreezingTemperatures` such that it returns an array containing the freezing temperatures (the temperatures that are below 0). Use an arrow function (implicit return is optional).

Strings (recap + advanced)

Let's take a look at some additional string **methods**:

String .trim()

This one is especially useful when working with the DOM (we'll learn about the DOM later, see slides 001, JS course structure) and you expect the user to enter some text. Users will sometimes, by accident, enter an empty space character at the beginning or the end of a text box.

Say, for example, you ask the user for their name, and they enter " Sam" instead of "Sam". This could cause issues in some scenarios (for example email addresses).

The solution for that is using the .trim() method which removes all leading (at the beginning) and trailing (at the end) space characters.

```
const name = " Sam Blue ";  
name.trim(); // "Sam Blue"
```



[string.trim\(\)](#) on MDN

String.startsWith() and .endsWith()

The `.startsWith(substring)` method returns `true` when the `substring` is found at the **beginning** of the string, and `false` otherwise. The `.endsWith(substring)` works the same but for the **end** of the string.

```
const sentence = "Hello there. Welcome!";
```

```
sentence.startsWith("H"); // true  
sentence.startsWith("Hello"); // true  
sentence.startsWith("Hey"); // false  
sentence.startsWith("Sam"); // false
```

```
sentence.endsWith("."); // false  
sentence.endsWith("!"); // true  
sentence.endsWith("Welcome!"); // true  
sentence.endsWith("Welcome"); // false
```



[string.startsWith\(\)](#) on MDN



[string.endsWith\(\)](#) on MDN

String.includes(substring)

The `.includes(substring)` method returns `true` when the `substring` can be found *anywhere* in the string, and `false` otherwise.

```
const sentence = "Hello there. Welcome!";
```

```
sentence.includes("there"); // true  
sentence.includes("W"); // true  
sentence.includes("Hello"); // true  
sentence.includes("Hey"); // false  
sentence.includes("Sam"); // false  
sentence.includes("."); // true  
sentence.includes("!"); // true  
sentence.includes("Welcome"); // true
```



[string.includes\(\)](#) on MDN

String.split(separator)

The `.split(separator)` method divides the string into an array by splitting it with the `separator` you provide. For example:

```
let apps = "Calculator,Phone,Contacts";  
let appsArray = apps.split(",");  
console.log(appsArray); // ["Calculator", "Phone", "Contacts"]
```

This is especially useful to convert from CSV (Comma Separated Values) back to an array. Reminder that the opposite of `String.split(separator)` would be `Array.join(glue)`.



[String.split\(\)](#) on MDN

String.replace(search, replace)

The `.replace(search, replace)` method returns a new string where the **first** occurrence of the `search` parameter you provide is replaced by the `replace` parameter, for example:

```
const message = "You are welcome.";
message.replace(" ", "_"); // "You_are welcome";
console.log(message); // "You are welcome" (original string is not changed)
```

In this example, we search for the `" "` (space character) and replace it with an `"_"` (underscore character). Notice how it only replaces the **first** match.

String.replace(search, replace)

If you'd like to replace all the occurrences, then you can use the `.replaceAll()` method explained below. It is also possible to pass a regular expression as a first parameter (which can be used to match more than one item. Though, generally, using `.replaceAll()` is easier).



[String.replace\(\)](#) on MDN

String.replaceAll(search, replace)

This method works like the one above, except that it will replace **all occurrences**.

```
const message = "You are welcome.";
message.replaceAll(" ", "_"); // "You_are_welcome";
console.log(message); // "You are welcome" (original string is not changed)
```



[String.replaceAll\(\)](#) on MDN

Practice / Classroom coding

strings01.js: Complete the function `getMessage` such that it returns the `message` it receives as a parameter. The `message` should always end with a full stop (.) Check the sample output to better understand the requirements.

strings02.js: Complete the function `getUnit` such that it returns the unit of temperature measurement used in the `text` it receives.

- It should return `"Celsius"` when `°C` is found in the string.
- It should return `"Fahrenheit"` when `°F` is found in the string.
- It should return `"N/A"` (Not Applicable) in all other cases.

To avoid small typos, make sure to copy the strings (such as Celsius and the degree symbol) from the challenge description.

Practice / Classroom coding

strings03.js: In computer science, a *slug* is a string used to identify a certain item. Oftentimes, this *slug* is used in the URL for Search Engine Optimization and better user experience. Let's say you've got a product called: "Easy assembly dining table". We cannot use this name in the URL bar because it contains spaces (it won't look nice, for example <https://example.com/item/Easy assembly dining table>). This is why we use a slug that looks like this: [easy-assembly-dining-table](https://example.com/item/easy-assembly-dining-table) so the URL becomes: <https://example.com/item/easy-assembly-dining-table>.

Complete the function `getSlug()` such that it returns the slug based on the text it receives, following the rules below:

- The slug should not be more than 15 characters. When there are more than 15 characters, ignore everything after the 15th character.
- The slug should always be in lower case.
- Space characters should be replaced by dashes (-).

Array (recap + advanced)

Here's a neat trick: Let's say you have an array of users, and you'd like to get the name of each user separated by a comma. Like a CSV (Comma Separated Values) export. How would you do that?

You already know the 2 methods that you need for such operations:

First, you start with a `.map()` call to extract the `name` from the `users` array , and then you use `.join()` to join the array elements into one string.

Array (recap + advanced)

```
const users = [{  
  id: 1,  
  name: "Sam Doe"  
}, {  
  id: 2,  
  name: "Alex Blue"  
}];
```

```
const userNamesArray = users.map(user => user.name);  
console.log(userNamesArray); // ["Sam Doe", "Alex Blue"];
```

```
const csv = userNamesArray.join(", ");  
console.log(csv); // "Sam Doe, Alex Blue"
```

You can also **chain** these 2 commands and write it on one line:

```
const csv = users.map(user => user.name).join(", ");  
console.log(csv); // "Sam Doe, Alex Blue"
```

Pretty neat right? ✨

Less used array methods

Array.every(callback)

The Array `.every(callback)` method returns `true` when **every** item in the array satisfies the condition provided in the callback.

Here's an example:

```
const numbers = [15, 10, 20];
```

```
const allAbove10 = numbers.every(number => number >= 10); // true
const allAbove15 = numbers.every(number => number >= 15); // false
```

Notice how `allAbove10` evaluates to `true` because **every** item in the `numbers` array returned `true` for the condition `number >= 10`. So, the callback `number => number >= 10` is being called for every item in the array. And, if all the callbacks returned `true`, then the `.every()` method will return `true`. Otherwise, it will return `false` (if at least one of the callbacks returned `false`).

This is why `allAbove15` evaluates to `false` because not all the items inside the `items` array satisfied the condition `number >= 15`. One of them (`number = 10`) returned `false` from the callback. So, the `.every()` method returns `false`.

Array.some(callback)

Similarly, the Array `.some(callback)` method returns `true` when **at least one** item in the array satisfies the condition provided in the callback.

```
const numbers = [15, 10, 20];
```

```
const someOver18 = numbers.some(number => number >= 18); // true  
const someUnder10 = numbers.some(number => number < 10); // false
```

Notice how `someOver18` evaluated to `true` because **at least one** of the items in the `numbers` array returned `true` for the condition `number >= 18`.

Whereas `someUnder10` evaluated to `false` because **none** of the items in the `numbers` array returned `true`.



[Array.some\(\)](#) on MDN

Deleting items

Say you've got an array and you'd like to empty it, you can do that by setting its length to 0:

```
const items = ["Pen", "Paper"];  
items.length = 0;
```

```
console.log(items); // []
```

This works even though we're using `const` because we're not re-assigning `items` but rather changing its length to 0 which ends up removing all the items inside of it. The `const` here guarantees that we will always have an array (but its content can change).

Array.splice()

You can also delete specific items from an array using the `splice(start[, deleteCount])` method.

Do not confuse `splice` with another method called `slice`.

Did you notice the `[, deleteCount]` syntax? You will often see this syntax in documentation, which means that the `deleteCount` parameter is **optional**.

The `.splice(start[, deleteCount])` method removes items from the array starting from the `start` index that you specify. If no `deleteCount` is provided, it will remove all the remaining items as of the `start` index.

When you specify a `deleteCount`, then it will remove as many items as you provided in the `deleteCount` from the `start` index.

The definition is more complicated than the example, so we recommend you take a look at the examples below:

- To delete the 1st element of an array, you call `.splice(0, 1)`.
- To delete 3 elements starting from the 2nd position, you call `.splice(1, 3)`.
- If you call `.splice(1)`, then it will remove all the items starting from the 2nd position (index 1).

Array.splice()

Try to always specify the 2nd argument for splice to avoid unexpectedly removing more items than necessary.

```
const items = ["Pen", "Paper", "Staples"];
const deletedItem = items.splice(0, 1); // removes one element at
index 0
console.log(deletedItem); // ["Pen"]
```

```
console.log(items); // ["Paper", "Staples"]
```

Notice how the `.splice` method **returns** an array of the element(s) removed.



[Array.splice\(\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) on MDN

Practice / Classroom coding

array01.js: Complete the function `shouldAdjustGrades` such that it returns `true` when the grades need to be increased.

Grades should be increased when at least one of them is failing (10 and above are passing).

array02.js: Complete the function `shouldCancelExam` such that it returns `true` when the exam needs to be canceled. An exam must be canceled if all the students' grades were very high (18 and above).

array03.js: Complete the function `resetApps` such that it empties the `apps` array it receives as a parameter.

array04.js: Complete the function `removeFirstApp` such that it removes the first element of the `apps` array it receives and returns the new array (which should not contain the item that was removed).

array05.js: Complete the function `removeSecondApp` such that it removes the second element from the `apps` array it receives and returns the new array (which should not contain the item that was removed).

What is reduce?

The `reduce()` method is a little bit complicated but we'll break it down step by step. Don't worry if you don't understand it the first time.

The goal of the `reduce()` method is to calculate a single value from an array. In other terms, you **reduce** an array into a single value.

Reduce example: sum

We can **reduce** the array `[5, 10, 5]` to the number 20.

We can **reduce** the array `[2, 4, 3]` to the number 9.

You may have noticed that both of these examples have summed up the items in the array. $5 + 10 + 5 = 20$ and $2 + 4 + 3 = 9$.

This is one of the most common use cases of `reduce()`, which is summing the array items.

What is reduce?

Reduce example: multiplication

Another example is multiplication. For example:

We can **reduce** the array [10, 2, 2] to the number 40 ($10 * 2 * 2 = 40$).

So is the reduce method a sum or a multiplication?

It's neither. That's because the `reduce()` method accepts the **reducer** which is a callback that you have to write. That callback can be sum, multiplication, or some other logic that you may think of.

So reduce is a generic function that will reduce an array into a single value. The way it will reduce that array into a single value is configurable by you, the developer. You can configure that in the `reducer` callback.

Array reduce: sum

Time to take a look at the code! In this lesson, we'll focus on one use case of reduce: calculating the sum. Thus, reducing an array of numbers into its sum.

Assuming the grades array below:

```
const grades = [10, 15, 5];
```

Here's how we can calculate its sum with reduce:

```
const sum = grades.reduce((total, current) => {  
  return total + current;  
}, 0);
```

There's a lot of things happening here, let's break them down:

1. We call the `.reduce()` method on the `grades` array.
2. We assign the result of `grades.reduce()` to a new variable called `sum`.
3. The reduce method is taking 2 arguments: The reducer and the initial value.

Reducer

The reducer in this example is:

```
(total, current) => {  
  return total + current;  
}
```

This is the callback that is applied for every item in the array, however, this callback takes 2 parameters: `total` and `current`.

The `total` is always referring to the last computed value by the reduce function. You may often see this called as `accumulator` in documentation which is a more correct name. And the `current` is referring to a single item in the array.

Reducer

Let's see what this means by showing the value of `total` and `current` for every step:

```
// code that we run
const grades = [10, 15, 5];

const sum = grades.reduce((total, current) => {
  return total + current;
}, 0);
```

- 1.The first time the callback runs, `total` is assigned 0 (because of the initial value that we'll explain in a bit) and `current` will be assigned to the first item of the array. So `total = 0` and `current = 10`.
- 2.Then we return `total + current` which is `0 + 10 = 10`. Now the new value of `total` becomes 10.
- 3.The callback runs the second time and this time `current = 15` (second item of the array) and `total = 10`. We return `total + current` which is `10 + 15 = 25`. The current value of `total` becomes 25.
- 4.The callback runs the third time and this time `current = 5` (third item of the array) and `total = 25`. We return `total + current` which is `25 + 5 = 30`.
- 5.There are no more items in the array, so the result of the reduce is the final value of `total` which is 30. Thus the sum is 30.

Initial Value

The `.reduce()` method accepts 2 arguments: `reducer` and `initialValue` (not to be confused by the 2 parameters of the `reducer` which are `total` and `current`). We explained the `reducer` above. The `initialValue` is the value we give to the `total` (or `accumulator`) the first time the callback runs.

Passing `0` as `initialValue` is the same as declaring `let sum = 0` when using `.forEach()` to calculate the sum.

It's the starting value that we use to be able to calculate the sum.

JavaScript will automatically take your `initialValue` and pass it to the `total` argument in the `reducer` the first time that callback runs.

So is the `initialValue` always `0`? When calculating the `sum` yes. We'll discuss other values in the next lesson.

Array reduce: multiplication

We'll explore another example for array reduce which is **multiplication**. Let's say we've got the following numbers and we'd like to multiply them all:

```
const numbers = [5, 2, 10];
```

We can use `.reduce()` here because we're reducing the entire array into a single number (which is the multiplication of all these numbers).

```
const result = numbers.reduce((total, current) => {  
  return total * current;  
}, 1);  
console.log(result); // 100
```

Starting value for multiplication

Before we explain the code step by step, let's talk about the **startingValue** which has a value of **1** here.

When doing multiplication, we can't have a starting value of **0**. That's because any number multiplied by 0 will result in 0. $5 * 0 = 0$. We need a number that is neutral in multiplication, and that number is **1** because any number multiplied by 1 will be that same number. For example, $5 * 1 = 5$.

This is why in multiplication we use a starting value of **1** and in sum, we use a starting value of **0**.

Step by step explanation

Going back to the code, here's how it runs step by step:

```
const numbers = [5, 2, 10];
```

```
const result = numbers.reduce((total, current) => {  
  return total * current;  
}, 1);
```

1. The first time `.reduce()` callback runs, `total` will have a value of `1` (coming from the starting value) and `current` will have a value of `5` (which is the first item of the array).
2. Then we return `total * current` which is $5 * 1 = 5$ so the next time the callback runs, `total` will have a value of `5`.
3. The second time the callback runs, `total` is `5` and `current` is `2` (second item of the array). We compute $5 * 2 = 10$. We return `10`.
4. The third time the callback runs, `total` is `10` and `current` is `10` (third item of the array). We compute $10 * 10 = 100$. We return `100`.
5. The result of the `.reduce()` is `100` which is stored in the variable `result`.

Common mistakes

When it comes to `.reduce()`, there are 3 common mistakes:

Syntax errors

Due to the number of parentheses and curly braces, it can get quite messy. Follow the same approach we recommended with `.forEach()`. That is re-writing the code from scratch and writing it on pen and paper.

Forgetting to return

Forgetting to return will lead to `undefined` values which will most likely end up giving you a result of `NaN`. We haven't seen `NaN` but it means **Not a Number**.

Make sure that you return from inside the `.reduce()` callback.

Wrong initialValue

The last common mistake is forgetting the `initialValue` or providing a wrong `initialValue`.

If you provide an `initialValue` of `0` for a multiplication, you will end up with a `0` at the end which should be a cue that the `initialValue` was wrong.

Chapter recap

- The `reduce()` method is used to calculate a single value from an array.
- In other terms, the `reduce()` method **reduces** an array into a single value.
- The most common use cases of reduce (when working with arrays of numbers) are sum & multiplication.
- The `reduce()` method takes a reducer which allows you to configure the logic of how the array will be reduced into a single number.
- The `reduce()` method takes 2 parameters: `reducer` and `initialValue`. `.reduce(reducer, initialValue)`.
- The `initialValue` is always 0 for sum.
- The reducer is a callback function that receives 2 arguments: `total` and `current`.
- The `total` (also called `accumulator`) keeps track of the result of the reduce method. For example, when calculating the sum, it keeps track of the sum, step by step.
- The `current` represents one item of the array.
- The `reducer` is called for every item in the array.
- For multiplication, we use an `initialValue` of 1.
- Reduce common mistakes:
 1. Syntax errors
 2. Forgetting to return
 3. Wrong `initialValue`

Practice / Classroom coding

reduce01.js: All you have to do is visualize in the console the value of `total`, `current`, and finally the `sum`. Notice how:

- the value of `total` starts at 0 (which is given by the `initialValue`)
- the value of `current` a single item of the array for every iteration.
- the value of `total` is the result of the last run of the reducer (the ongoing `sum`)

Feel free to change the numbers in the `grades` array and check the output in the console to better understand how `.reduce()` works.

reduce02.js: Calculate the sum of the `grades` array and store it in a variable called `sum`.

reduce03.js: Complete the function such that it returns the sum of the `numbers` it receives as a parameter.

reduce04.js: Complete the function `multiplyNumbers` such that it multiplies every number from the `numbers` parameter it receives.

Optional Chaining

Enter optional chaining. It lets us take an example code and look into the following:

```
// assuming object `user`
```

```
const name = user.details?.name?.firstName;
```

So, yeah! All the code can be replaced with this single line. Notice the `?.` operator after `user.details` and then after `user.details?.name`.

This is called optional chaining which allows you to access a property deep within an object without risking an error if one of the properties is `null` or `undefined`.

In case one of the properties is `null` or `undefined`, then the `?.` will short-circuit to `undefined`.

This means that it will stop reading the property you asked it to read and will result in `undefined`.

Optional Chaining

Let's take a look at some examples and see the result of optional chaining:

```
const user = {  
  details: {  
    name: {  
      firstName: "Sam"  
    }  
  },  
  data: null  
}
```

```
user.details?.name?.firstName; // "Sam"  
user.data?.id; // undefined  
user.children?.names; // undefined  
user.details?.parent?.firstName; // undefined
```



Optional chaining usage with arrays

Assuming the code below, where the key `temperatures` might be `undefined`:

```
const data = {  
  temperatures: [-3, 14, 4]  
}
```

```
let firstValue = undefined;  
if (data.temperatures) {  
  firstValue = data.temperatures[0];  
}
```

We can refactor it into:

```
const firstValue = data.temperatures?.[0];
```

Notice how we used `?.` in front of the `[0]` to access the first item of the array. The benefit of optional chaining here is that if `data.temperatures` returned `null` or `undefined`, your code won't break. It will short-circuit and return `undefined`. Which is why we were able to get rid of the `if` condition.

Optional chaining usage with functions

Similarly, we can use optional chaining to call functions by using the `?.` operator. Here's the example **before** optional chaining:

```
const person = {  
  age: 43,  
  name: "Sam"  
};
```

```
let upperCasedName = person.name; // might be undefined  
if (person.name) {  
  upperCasedName = person.name.toUpperCase();  
}
```

Which we can refactor into:

```
const upperCasedName = person.name?.toUpperCase();
```

If at any point the code above is not clear, wait until the challenges, and try running your code without the `if` condition and without optional chaining and notice that it fails with errors such as *Cannot read property 'toUpperCase' of undefined*.

Optional chaining helps you avoid these errors by returning `undefined`.

Do not overuse optional chaining

You might be tempted to start using optional chaining instead of every **dot notation** but you shouldn't. Overusing it may lead to unexpected bugs and poor code quality.

You can think of it as the following: When I (or other programmers) see `?.` in the code, it means that there's a moderate chance that the value returns `undefined`. In turn, this means that we should be handling the case when it returns `undefined`.

For example, the below code does **not** handle the `undefined` case:

```
const sum = values => {  
  return values?.[0] + values?.[1];  
}
```

```
sum([2, 3]); // 5  
sum([]); // NaN
```


As you can see, even though our code doesn't break, `sum([])` returns `NaN` which in turn might cause unexpected issues later on.

The above code can be fixed by adding an `if` condition that checks whether we have 2 entries in the `values` array, otherwise we return `0`.

No optional chaining for assignment

Optional chaining is *only* used for **reading**. It **cannot** be used for assignment.

This means that the code below is **invalid** and will throw a syntax error:

```
//  Syntax Error  
settings?.theme = "dark";
```

Chapter Recap

- Optional chaining allows you to access a property deep within an object without risking an error if one of the properties is `null` or `undefined`.
- In case one of the properties is `null` or `undefined`, then the `?.` will short-circuit to `undefined`.
- You **cannot** use optional chaining on an object that may not exist. The object **has** to exist. Optional chaining is only used to access a property that may or may not exist.
- Optional chaining can be used for arrays. The syntax is `?.[index]`
- Optional chaining can be used for functions. The syntax is `?.functionName()`
- Optional chaining **cannot** be used for assignment. It's *only* used for **reading**.

Practice / Classroom coding

- **optional/optional01.js**: Complete the function `getFullName` such that it returns the full name from the user object when it exists. Otherwise, it should return `undefined`.
- **optional02.js**: Re-write the `getPaymentValue` function without using `if` conditions.
- **optional03.js**: Re-write the `getFirstGrade` function without using `if` conditions.
- **optional04.js**: Complete the function `getFullName` such that it returns the full name **in lower case** from the user object when it exists. Otherwise, it should return `undefined`.

Supermini Project 03

In this supermini project 03, suppose we have a form that accepts a **promo code**. The promo code is only considered valid when it's between 5 and 10 characters (inclusive). This means 5, 6, 7, 8, 9, and 10 characters are considered valid, but everything else is not.

Complete the function **isPromoCodeValid** such that it returns `true` when the promo code is valid and `false` otherwise.

Supermini Project 04

In this supermini project 04, complete the 2 functions in `shopping.js` that will complete this shopping list app.

1. `addItem` should add the `item` it receives as a parameter to the `items` array.
2. `exportLowerCasedCSV` should return a string containing all the items in *lower case* and separated by a comma and a space character.

Supermini Project 05

In this supermini project **classroom app**, serving as a recap for the arrays of numbers, you will need to:

- 1.implement submitting a grade (it should add the `grade` to the `grades` array)
- 2.show the number of tests taken.
- 3.show the first submitted grade (first submitted, not the highest).
- 4.show the last submitted grade (last submitted, not the lowest).
- 5.show the sum of all the grades
- 6.show the average grade
- 7.show the grades raised by 2 (every grade increased by 2 points)