Course syllabus

This course is the second of a series that aims to help you learn more about web development.

In this course, you will explore the following:

**Module 1: Introduction to JavaScript**

In this module, you are introduced to JavaScript. You'll learn why JavaScript is so integral to software development. And you'll get an overview of how to write JavaScript code inside the browser. Furthermore, you will learn about the most common operators as well as conditional statements and loops.

After completing this module, you will be able to:

* Explain the importance of JavaScript in software development
* Demonstrate how to write JavaScript code inside the browser
* Demonstrate how to write basic JavaScript code
* List common operators, conditional statements and loops
* Demonstrate how to use variables and output their value in the console

**Module 2: The building blocks of a program**

Here you'll learn how to use objects, arrays and functions. In addition, you will learn about the most common built-in methods, and the difference between undefined, null and empty strings. And you'll explore both error handling and defensive programming.

After completing this module, you will be able to:

* Build and use objects, arrays, and functions
* List some common built-in methods on built-in objects
* Describe handling bugs and errors using try, catch, throw, and defensive programming
* Explain the difference between undefined, null, and empty strings
* Demonstrate how to write basic code using arrays, objects and functions

**Module 3: Programming paradigms**

This module is about functional programming and the object oriented programming paradigm. You will learn what scope is in JavaScript. You'll explore the differences between var, let and const. And you'll learn how to use classes and inheritance in object oriented programming. Additionally, you'll explore how to use write JavaScript using modern features like spread and rest. You will build code that can manipulate the DOM and handle events. And you will use JSON in JavaScript.

After completing this module, you will be able to:

* Outline the tenets of the functional programming and object oriented programming paradigm
* Describe how scope works in JavaScript
* List the differences between var, let, and const
* Use classes and inheritance in OOP in JavaScript
* Write JavaScript code using more modern features like spread, rest, template strings and modules
* Build code that manipulates the DOM and handles events
* Use JSON in JavaScript

**Module 4: Testing and compatibility**

Here you will learn about Node.js and npm. And you will explore how to install npm packages and how to work with package.json. Furthermore, you will learn about testing in JavaScript and you'll code a simple unit test in Jest.

After completing this module, you will be able to:

* Describe Node.js and npm
* Explain how to install npm packages
* Describe how to work with package.json
* Explain the process of testing in JavaScript
* List the three most prevalent kinds of testing
* Demonstrate how to code a simple unit test in Jest

**Module 5: Graded assessment**

In the final module, you'll learn about the graded assessment. After you complete the individual units in this module, you'll synthesize the skills you gained from the course to create code for the "Little lemon receipt maker ".

You'll also have to opportunity to reflect on the course content and the learning path that lies ahead.

# How to Position Yourself for a New Career

You are well on your way to becoming a software developer.

You took the most important first step: you started.

While this specialization on Coursera will make you into a well-rounded junior developer, you are basically just getting started.

Here are some proven tips to make the transition to your new career as smooth as possible.

## **Be persistent**

Succeeding in your career efforts is not easy. Luckily, it's not too hard either. Consider this new endeavor of learning to code a part of your everyday life.

Make it as much of a routine as possible. Hopefully, it will work like this:

* You wake up,
* You brush your teeth,
* You run some errands,
* And then you write and learn to code.

Obviously, there are things like your school obligations, or your day job, or other places you need to be and things you need to do.

However, if you don't code regularly - preferably on a daily basis - your progress will be slower. Try to set aside some time to consistently code and learn every day. Persistence is key.

## **Start building simple apps today**

Don't wait until you "learn enough". There's always more to learn, and it's best to get started with any kind of a simple project right now.

Even just taking the code from this specialization and tinkering with it will do wonders for your confidence and the speed at which you acquire new knowledge.

Also, the more you practice, the better you'll retain what you've learned.

Having your own projects that you can showcase to others - no matter how small or straightforward shows a track record and dedication. This is something that your future employer might be impressed with, so start today.

## **Set up a GitHub account**

Since we're on the topic of personal projects, head on over to [GitHub](https://github.com/) and set up your developer profile right away. It's essential to have an account there since you can keep all your projects in a single location that you can access from any computer.

You can almost think of your GitHub account as an additional brain power. No matter how long ago, whatever you've worked on will remain there, waiting for you to peek into and re-familiarise yourself with.

## **Pair program**

Try to find someone at your level or perhaps slightly more knowledgeable than you and ask them to set up a recurring pair programming session.

This works nicely because having a pair programming partner can speed up your learning. You also have someone to be accountable to.

## **Start a coding blog**

Technical communication is important for developers, and just like anything else, you get better with practice.

Starting a coding blog will work the same as having a GitHub account, with some the additional benefits:

* It shows even more dedication - and this increases your chances of getting hired
* It helps you experiment with different technologies
* Setting up your own website is practical learning in its own right and one more project to add to your CV

## **Collaborate on open source projects**

Even if you are just starting out and are really struggling to get into this field, you can still be a valuable contributor to open-source projects.

There are so many open-source projects that are in demand for all kinds of contributors.

Even contributing to a project by fixing some typos in documentation files is a nice start to getting more involved and putting yourself out there.

## **Get a certificate**

Getting certified is always a good thing. The fact that you're reading this lesson right now confirms that you're on your way to receiving a certificate of completion from Coursera!

## **Keep a positive attitude**

As with anything worth doing, you might sometimes get tired, not understanding how something works, and perhaps even feel like giving up.

Remember to stay consistent.

There are always ups and downs in life, but sometimes it's worth it to think of all the things you've achieved so far and use that as motivation to keep at it.

## **Never stop learning**

There's always more to learn in IT, and that's probably the best thing about it. It's the very thing that makes it fun and provides an opportunity for each developer to get ahead in their career.

# How to uncover job opportunities

Learning how to program in JavaScript helps you prepare for a wide range of job opportunities. This is in part because it expands the possibilities of what you can build as a developer.

JavaScript is one of the most in-demand programming languages, as it is used in nearly all active websites. Its versatility enables developers to use popular libraries, plugins, and frameworks such as React, which improves efficiency and productivity.

With all flexibility that programming in JavaScript brings, it is no surprise that there are different careers that you might want to follow. However, no matter what career path you choose, you will always want to learn other technologies like HTML, CSS, React, Node.js, or Python, so you are more marketable.

Let’s cover a few of the most common roles you can get if you know how to program in JavaScript.

**Mobile Developer** There has been a constant increase in the demand for mobile developers as mobile devices’ use for accessing the internet has been on the rise. Mobile developers specialize in building apps for platforms like Google’s Android and Apple’s iOS. Many developers choose to use React Native, which allows them to build one application using JavaScript that works on both Android and iOS devices. Mobile developers work with UX and UI designers and use React’s UI capabilities to implement functionalities that customers will use. Mobile developers also make sure that the front-end and the back-end of the applications work seamlessly. Other skills that mobile developers may have include HTML, CSS, Java, Kotlin, Objective-C, C++, C#, among others.

**Front-End Developer** As the name implies, front-end developers build the user-facing parts of websites and apps. They work closely with designers to implement visual and interactive elements through coding, using HTML, CSS, and of course, JavaScript. They also use libraries and frameworks like React to save time and make their work more efficient. Front-end developers may also be responsible for making sure the final user has a good user experience and that websites and apps behave as expected and are free of errors and bugs.

**Back-End Developer** Back-end developers work on the back-end of websites and applications. They can use JavaScript with Node.js to develop back-end functionalities. Among these functionalities are streaming and chat-based applications, as well as JSON APIs and serverless functions. Back-end developers possess additional skills, including professional working knowledge of Python, APIs, cloud infrastructure, and database.

**Full-Stack Developer** As you might guess, a full-stack developer works with both the front- and back-end of building websites and apps. So, these professionals combine the skills in these two areas. They can use frameworks like React to work on the front-end and Node.js to work on the back-end. They also apply skills in addition to JavaScript to build websites and apps.

**Your professional journey ahead**

As you embark on the exciting career as a professional developer, you will realize that you will progressively expand your skills to include a wide range of technologies and programming languages besides JavaScript.

If you want to have an idea of the opportunities available in these areas, you might want to check your favorite job search website or app and look for jobs related to JavaScript. You will learn that there is no shortage of opportunities. And as you start reading more about these roles, you will find that JavaScript is often only one of the competencies that employers are looking for.

But don’t worry, as you advance in your studies and in your career, you will further develop your skills to focus on the professional path you want to follow.

Good luck on your professional journey!

# Writing your first Javascript code

In this reading, you'll learn about comments in JavaScript. Additionally, you'll learn about the semi-colon in JavaScript: what it does and why it is used. You will then download a browser if you don't have one installed and run your first piece of JavaScript using the Console.

## Comments in JavaScript

I've chosen comments as the starting point for two reasons:

1. Their syntax - the way comments are written is easy to understand.
2. Writing comments can empower you as a developer.

First, I'll explain the syntax, and after that, I'll discuss why being able to write comments is so empowering.

### Comments in JavaScript: the syntax

There are two varieties of comments in JavaScript:

1. Single-line comments
2. Multi-line comments

A single-line comment is created when you add two forward-slash characters one after the other, without spaces.

1

// this is a comment!





Anything that follows a single-line comment in JavaScript is ignored by the browser.

This means that, essentially, you can write any kind of text, code, characters, emojis, whatever - and the browser will ignore it.

A multi-line comment, as its name says, spans for several lines of code and is created with a forward slash and a star. For example:

4

5

6

7

a

multi-line

comment

\*/





You can also use the multi-line comment syntax on a single line of code, as follows:

1

/\* this is a multi-line comment on a single line \*/





## Why writing comments is empowering

In this course, it is assumed that you've never written a single line of JavaScript code.

With this assumption in mind, consider the effects of what you've just learned, that is, the effects of learning how to write comments in JavaScript:

1. You can now freely express your ideas about any code that you write.
2. You can add comments to any code that already exists.
3. Those comments can be intended for your future self, or for colleagues on your development team.

So, comments are empowering because they facilitate communication with your future self or with your team members, allowing you to ask questions about the code, mark the code as "to do", or as "to improve", or just simply explain what a given piece of code does.

Additionally, you can even comment out some working code in a JavaScript file - to prevent it from running.

Effectively, comments allow you to "switch off" pieces of JavaScript code.

There can be many reasons for that:

1. Trying to understand how a given piece of code works.
2. Testing different solutions to a coding problem - while not having to delete existing code.
3. Debugging - trying to pin-point why your code is broken or behaving unprediticably.

## The semi-colon in JavaScript

In the English language, the fullstop or period - the **.** character - is used to separate thoughts into sentences.

By clearly separating thoughts with the fullstop, you avoid being misunderstood.

In JavaScript, the semi-colon - the **;** character - has a similar purpose: it is used to clearly delimit parts of the code from some other parts of the code.

### Automatic Semi-Colon Insertion (ASI)

Interestingly, the browser has a feature known as "Automatic Semi-colon Insertion" - meaning, it does a pretty good job of "filling in the blanks" in case there is a missing semi-colon where there should be one.

Effectively, what that means for developers is that most of the time, it makes no difference if a semi-colon is added or not, since the browser is likely to figure it out anyway.

That's why some developers say that you shouldn't bother with adding semi-colons at all.

However, other developers argue that it's better to use it wherever it's needed - for the sake of clarity.

The truth is that most of the time, you can think of adding semi-colons in JavaScript as optional - and somewhat of a stylistic preference.

## A note on using the console in the developer tools in your browser

As already mentioned earlier on in this course, one of the reasons why JavaScript is so popular is because it's so approachable.

To get started with JavaScript, all you need is a browser. In this course I'll be using Google Chrome.

Once you've installed the browser and run it, right-click on the currently active web page and click the **Inspect** command on the right-click contextual menu.

This will open the Developer Tools and then you can click on the Console tab to open the console, or alternatively, pressing the **ESC** key will toggle on and off the console regardless of the currently active Developer Tools panel.

You can type any JavaScript command you like into the DevTools console.

## **If you need to type multiple lines of code before you run them, make sure to press the SHIFT + ENTER shortcut key to get onto the next line.**

Notice the distinction between pressing the **ENTER** key to run the JavaScript code you've typed, versus pressing the **SHIFT + ENTER** shortcut to move onto the next line of code (rather than running the code you've already typed up).

This is all that you need to get started writing JavaScript code!

In the upcoming lessons, feel free to follow along in either of two ways:

1. Using the VS Code editor and the Code Runner extension as previously described
2. Using the Chrome browser itself, and running the code inside the DevTools console as described in this reading

## Output a greeting into the console

Now that you know how to get to the Developer Tools' Console tab, let's now use it to run your first piece of real JavaScript code.

In Chrome, with the Developer Tools open, click into the empty space in the console tab, just to the right of the blue **>** character. You should see a blinking vertical line (also referred to as "the cursor"). The cursor indicates that you can type into the console.

If you type valid JavaScript code, it will be executed, meaning: it will be processed and it might result in some kind of output.

You'll now use the **console.log** function to output the words "Hello, World".

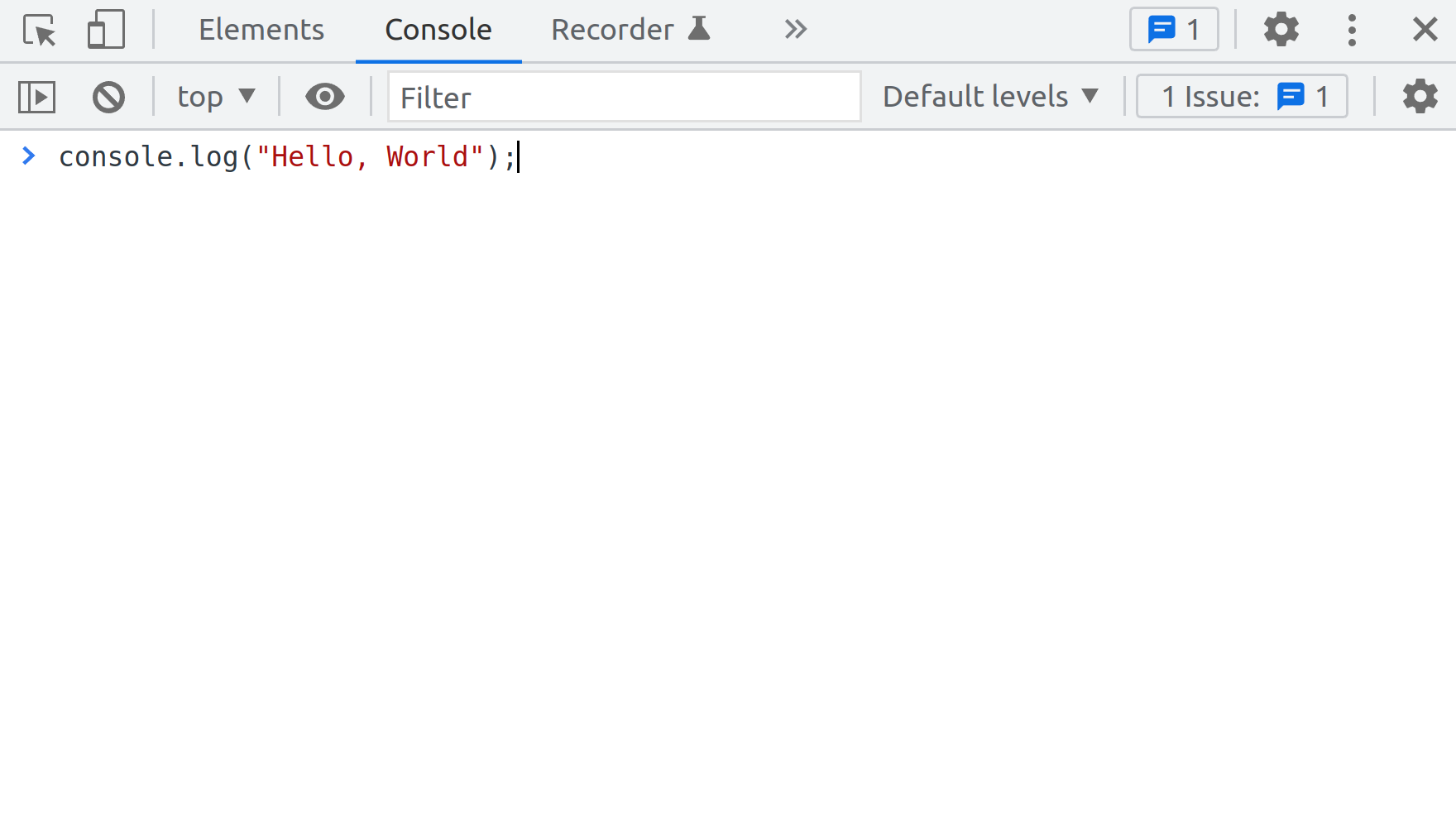
To do so, type the following command into the console:

1

console.log("Hello, World");







If you've done everything as instructed, the words "Hello, World" should be output in the console.

Here's another, more complex command, to show you that the **console.log** command comes with a number of tricks.

For example, did you know that you can style the output in the console?

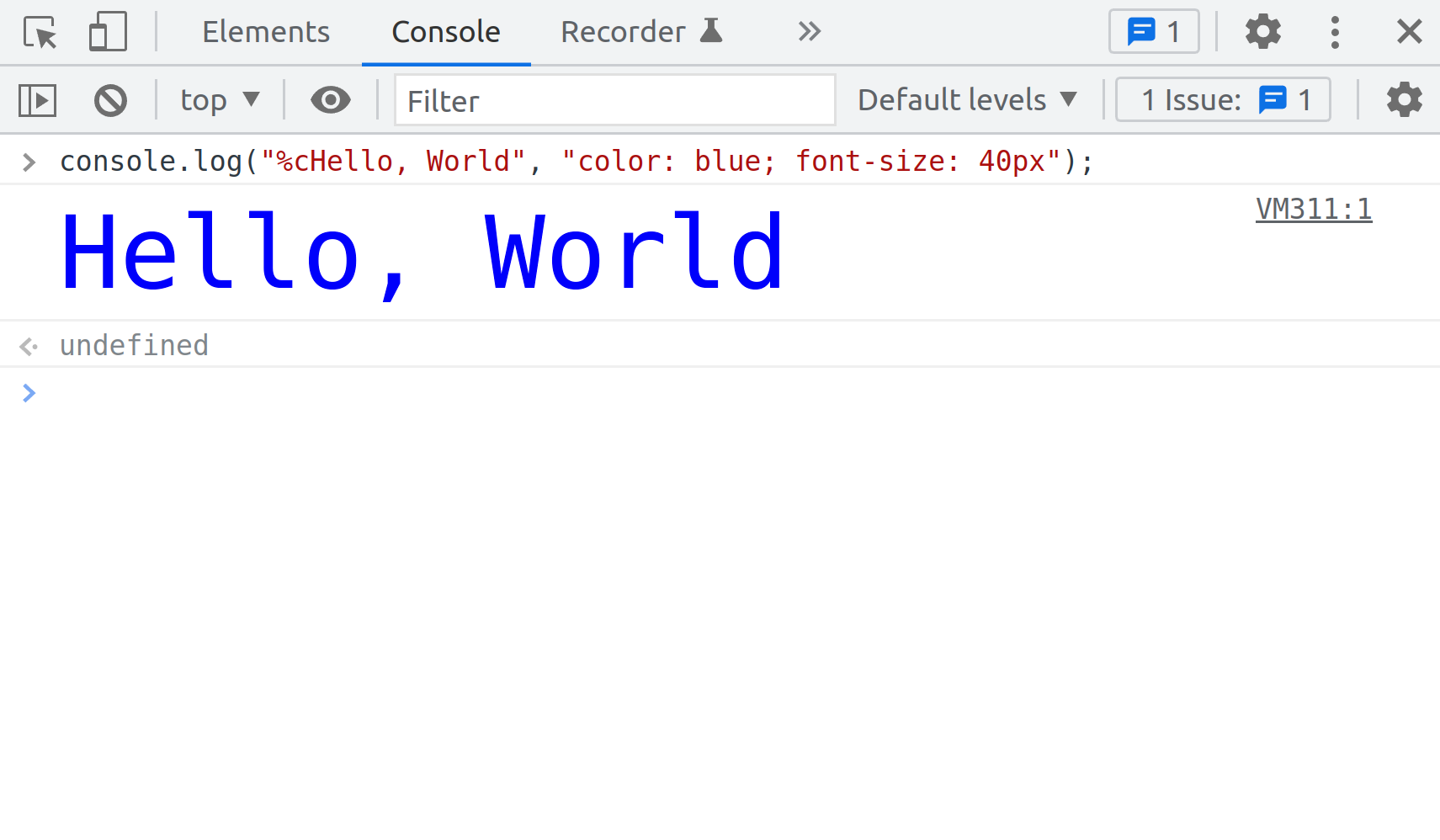
In this code snippet, there are a few additions: the font size is different and the color is blue:

1

console.log("%cHello, World", "color: blue; font-size: 40px");





console log with percentage c authors

If you copy-paste this piece of code, or perhaps, simply type it into the console, once you press the **ENTER** key to run it, you'll get the words "Hello, World" output to the console. This time, however, the color of the letters will be blue, and the font size will be 40px. So, you've just learned a nice trick with the console.

If you add the **%c** right after the **"** character, you can then style the console output by adding the **,** character after the second **"**, and then, inside another pair of **"** and **"** characters, use valid CSS code to style the words you want to output in the console.

The reason for showing you this little trick was to hopefully get you motivated to practice writing various words into the **console.log** command, and to use your CSS skills to change the styling of these words in the console output. That way, you might find it fun to practice this newly acquired skill - and learning and fun always go nicely together.

## Output multiple words into the console

To output multiple words into the console, you can join them using the **+** character, formally known as the "concatenation operator" when we're joining pieces of text, or the "addition operator", for performing the mathematical operation of adding two numbers.

Here is an example of joining three separate pieces of text: **console.log("Hello " + "there, " + "World")**. The output of this command will be: Hello there, World.

Here is an example of outputting three separate pieces of text using the , character instead:

**console.log("Hello ", "there, ", "World")**

The output of this command will still be: Hello there, World.

# Operators in depth

In this reading, you will learn about additional operators, operator precedence and operator associativity. I'll also provide you with some examples of logical operators.

## 1. Additional operators

* Logical AND operator: **&&**
* Logical OR operator: **||**
* Logical NOT operator: **!**
* The modulus operator: **%**
* The equality operator: **==**
* The strict equality operator: **===**
* The inequality operator: **!=**
* The strict inequality operator: **!==**
* The addition assignment operator: **+=**
* The concatenation assignment operator: **+=** (it's the same as the previous one - more on that later)

### The logical AND operator in JavaScript: &&

The **logical AND operator** is, for example, used to confirm if multiple comparisons will return true.

In JavaScript, this operator consists of two ampersand symbols together: **&&**.

Let's say you're tasked with coming up with some code that will check if the **currentTime** variable is between **9 a.m.** and **5 p.m**. The code needs to console.log **true** if **currentTime > 9** **and** if **currentTime < 17**.

Here's a solution:

var currentTime = 10;

console.log(currentTime > 9 && currentTime < 17);

How does this code work?

First, on line one, I set the **currentTime** variable, and assign the value of **10** to it.

Next, on line two I console log two comparisons:

**currentTime > 9**

**currentTime < 17**

I also use the **&&** logical operator to join the two comparisons.

Effectively, my code is interpretted as the following:

console.log(10 > 9 && 10 < 17);

The comparison of **10 > 9** will return **true**.

Also, the comparison of **10 < 17** will return **true**.

This means I can further re-write the line two of my solution as follows:

console.log(true && true);

In essence, this is how my code works.

Now, the question is, what will be the result of **console.log(true && true)**?

To understand the answer, you need to know the behavior of the **&&** logical operator.

The **&&** logical operator returns a single value: the boolean **true** or **false**, based on the following rules:

* It returns **true** if both the values on its right and on its left are evaluated to **true**
* It returns **false** in all the other instances

In other words:

**console.log(true && true)** will output: **true**

**console.log(true && false)** will output: **false**

**console.log(false && true)** will output: **false**

**console.log(false && false)** will output: **false**

### The logical OR operator in JavaScript: ||

The logical OR operator in JavaScript consists of two pipe symbols together: **||**.

It is used when you want to check if at least one of the given comparisons evaluates to **true**.

Consider the following task: You need to write a program in JavaScript which will return **true** if the value of the **currentTime** variable is not between **9** and **17**. Put differently, your code needs to console.log **true** if the value of the variable **currentTime** is either less than **9** or greater than **17**.

Here's a solution:

var currentTime = 7;

console.log(currentTime < 9 || currentTime > 17);

In line one of the code I assign the number **7** to the variable **currentTime**.

On line two, I console log the result of checking if either **currentTime < 9** or **currentTime > 17** will evaluate to **true**.

It's the same as this:

var currentTime = 7;

console.log(true || false);

Here are the rules of how the **||** operator evaluates given values:

**console.log(true || true)** will output: **true**

**console.log(true || false)** will output: **true**

**console.log(false || true)** will output: **true**

**console.log(false || false)** will output: **false**

The logical OR operator will always return **true**, except when both sides evaluate to **false**. In other words, for the logical OR operator to return **false**, the results of both comparisons **must** return false.

Going back to the example of checking if either **currentTime < 9** or **currentTime > 17**, this makes sense: the only time you will get **false** is when the value stored in the **currentTime** variable is greater than **9** and less then **17**.

### The logical NOT operator: !

In JavaScript, the logical NOT operator's symbol is the exclamation mark: **!**.

You can think of the **!** operator as a switch, which flips the evaluated boolean value from **true** to **false** and from **false** to **true**.

For example if I assign the boolean value of **true** to the **petHungry** variable:

**var petHungry = true;**

...then I can console log the fact that the pet is no longer hungry by using the **!** operator to flip the boolean value stored inside of the **petHungry** variable, like so:

**console.log('Feeding the pet'); console.log("Pet is hungry: ", !petHungry); console.log(petHungry);**

This is the output of the above code:

Pet is hungry: true

Feeding the pet

Pet is hungry: false

true

The reason for the changed output in the console is because you have flipped the value stored inside the **petHungry** variable, from **true** to **false**.

Notice, however, that the code on line five of the example above still outputs **true** - that's due to the fact that I didn't reassign the value of the **petHungry** variable.

Here's how I could permanently change the value stored in the **petHungry** variable from **true** to **false**:

var petHungry = true;

petHungry = !petHungry;

In this example, I first assign the value of **true** to the new variable of **petHungry**. Then, on line two, I assign the opposite value, the **!true** - read: not true - to the existing **petHungry** variable.

### The modulus operator: %

The modulus operator is another mathematical operator in JavaScript. It returns the remainder of division.

To demonstrate how it works, imagine that a small restaurant that has 4 chairs per table, and a total of 5 tables, suddenly receives 22 guests.

How many guests will not be able to sit down in the restaurant?

You can use the modulus operator to solve this.

console.log(22 % 5); // 2

The output is **2**, meaning, when I divide **22** and **5**, I get a **4**, and the remainder is **2**, meaning, there are **2** people who couldn't get a place in this restaurant.

### The equality operator, ==

The equality operator checks if two values are equal.

For example, this comparison returns **true**: **5 == 5**. Indeed, it is true that 5 is equal to 5.

Here's an example of the equality operator returning **false**: **5 == 6**. Indeed, it is true that 5 is not equal to 6.

Additionally, even if one of the compared values is of the number type, and the other is of the string type, the returned value is still **true**: **5 == "5"**.

This means that the equality operator compares only the values, but not the types.

### The strict equality operator, ===

The strict equality operator compares for both the values and the data types.

With the strict equality operator, comparing **5 === 5** still returns **true**. The values on each side of the strict equality operator have the same value and the same type. However, comparing **5 == "5"** now returns **false**, because the values are equal, but the data type is different.

### The inequality operator, !=

The inequality operator checks if two values are not the same, but it does not check against the difference in types.

For example, **5 != "5"** returns false, because it's false to claim that the number 5 is not equal to number 5, even though this other number is of the string data type.

### The strict inequality operator !==

For the strict inequality operator to return **false**, the compared values have to have the same value and the same data type.

For example, **5 !== 5** returns **false** because it is false to say that the number 5 is not of the same value and data type and another number 5.

However, comparing the number 5 and the string 5, using the strict inequality operator, returns **true**.

console.log(5 !== "5")

## 2. Using the + operators on strings and numbers

### Combining two strings using the + operator

The **+** operator, when used with number data type, adds those values together.

However, the **+** operator is also used to join string data type together.

For example:

"inter" + "net" // "internet"

"note" + "book" // "notebook"

If the **+** operator is used to join strings, then it is referred to as the concatenation operator, and you'll say that it's used to concatenate strings.

When used with numbers, the **+** operator is the **addition operator**, and when used with strings, the **+** operator is the **concatenation operator**.

### Combining strings and numbers using the + operator

But what happens when one combines a string and a number using the **+** operator?

Here's an example:

365 + " days" // "365 days"

12 + " months" // "12 months"

Here, JavaScript tries to help by converting the numbers to strings, and then **concatenating the number and the string together**, ending up with **a string value**.

The process of this "under-the-hood" conversion of values in JavaScript is referred to as "coercion". JavaScript coerces a number value to a string value - so that it can run the **+** operator on disparate data types.

The process of coercion can sometimes be a bit unexpected.

Consider the following example:

1

1 + "2"

What will be the result of **1 + "2"**?

Note that the value of **1** is of the number data type, and the value of **"2"** is of the string data type, and so JavaScript will coerce the number **1** to a string of **"1"**, and then concatenate it with the string of **"2"**, so the result is a string of **"12"**.

### The addition assignment operator, +=

The addition assignment operator is used when one wants to accumulate the values stored in a variable.

Here's an example: You are counting the number of overtime hours worked in a week.

You don't have to specify the type of work, you just want to count total hours.

You might code a program to track it, like this:

var mon = 1;

var tue = 2;

var wed = 1;

var thu = 2;

var fri = 3;

console.log(mon + tue + wed + thu + fri); // 9

You can simplify the above code by using the addition assignment operator, as follows:

var overtime = 1;

overtime += 2;

overtime += 1;

overtime += 2;

overtime += 3;

console.log(overtime); // 9

Using the addition assignment operator reduces the lines of your code.

### The concatenation assignment operator, +=

This operator's syntax is exactly the same as the addition assignment operator. The difference is in the data type used:

var longString = "";

longString += "Once";

longString += " upon";

longString += " a";

longString += " time";

longString += "...";

console.log(longString); // "Once upon a time..."

### Operator precedence and associativity

Operator precedence is a set of rules that determines which operator should be evaluated first.

Consider the following example:

1

1 \* 2 + 3

The result of the above code is 5, because the multiplication operator has precedence over the addition operator.

Operator associativity determines how the precedence works when the code uses operators with the same precedence.

There are two kinds:

* left-to-right associativity
* right-to-left associativity

For example, the assignment operator is right-to-left associative, while the greater than operator is left-to-right associative:

var num = 10; // the value on the right is assigned to the variable name on the left

5 > 4 > 3; // the 5 > 4 is evaluated first (to `true`), then true > 3 is evaluated to `false`, because the `true` value is coerced to `1`

# JavaScript improvements

In this reading, you will learn about the history of JavaScript and the importance of ECMA (European Computer Manufacturers Association) and ECMAScript.

JavaScript is a programming language that had humble beginnings.

It was built in only 10 days in 1995 by a single person, Brendan Eich, who was tasked with building a simple scripting language to be used in version 2 of the Netscape browser. It was initially called LiveScript, but since the Java language was so popular at the time, the name was changed to JavaScript - although Java and JavaScript are in no way related.

For the first few years, after it was built, JavaScript was a simple scripting language to add mouseover effects and other interactivity. Those effects were being added to webpages using the **<script>** HTML element.

Inside each of the script elements, there could be some JavaScript code. Due to the rule that HTML, CSS, and JavaScript must be backward compatible, even the most advanced code written in JavaScript today ends up being written between those script tags.

Over the years, JavaScript grew ever more powerful, and in recent times, it's continually touted as among the top three commonly used languages.

In 1996 Netscape made a deal with the organization known as ECMA (European Computer Manufacturers Association) to draft the specification of the JavaScript language, and in 1997 the first edition of the ECMAScript specification was published.

ECMA publishes this specification as the ECMA-262 standard.

You can think of a standard as an agreed-upon way of how things should work. Thus, ECMA-262 is a standard that specifies how the JavaScript language should work.

There have been 12 ECMA-262 updates - the first one was in 1997.

JavaScript as a language is not a completely separate, stand-alone entity. It only exists as an implementation. This implementation is known as a JavaScript engine.

Traditionally, the only environment in which it was possible to run a JavaScript engine, was the browser. More specifically, a JavaScript engine was just another building block of the browser. It was there to help a browser accomplish its users' goal of utilizing the internet for work, research, and play.

So, when developers write JavaScript code, they are using it to interact with a JavaScript engine. Put differently, developers write JavaScript code so that they can "talk to" a JavaScript engine.

Additionally, the JavaScript engine itself comes with different ways to interact with various other parts of the browser. These are known as Browser APIs.

Thus, the code that you write in the JavaScript programming language allows you to: 1. Interact with the JavaScript engine inside of the browser 2. Interact with other browser functionality that exists outside of the JavaScript engine, but is still inside the browser.

Although traditionally it was possible to interact with the JavaScript engine only inside of the browser, this all changed in 2009, when Node.js was built by Ryan Dahl.

He came up with a way to use a JavaScript engine as a stand-alone entity. Suddenly, it was possible to use JavaScript outside of the browser, as a separate program on the command line, or as a server-side environment.

Today, JavaScript is ubiquitous and is running in browsers, on servers, actually, on any device that can run a JavaScript engine.

Conditional examples

In this reading, you will learn when to use the **if else** statement and when to use the **switch** statement.

Both **if else** and **switch** are used to determine the program execution flow based on whether or not some conditions have been met.

This is why they are sometimes referred to as **flow control statements**. In other words, they control the flow of execution of your code, so that some code can be skipped, while other code can be executed.

At the heart of both flow control structures lies the evaluation of one or more conditions.

Generally, **if else** is better suited if there is a binary choice in the condition.

For example, in plain English: *if it's sunny, wear sunglasses. Otherwise, don't*.

In this case, using an if statement is an obvious choice.

When there are a smaller number of possible outcomes of truthy checks, it is still possible to use an **if else** statement, such as:

if(light == "green") {

    console.log("Drive")

} else if (light == "orange") {

    console.log("Get ready")

} else if (light == "red") {

    console.log("Dont' drive")

} else {

    //this block will run if no condition matches

    console.log("The car is not green, orange, or red");

}

However, if there are a lot of possible outcomes, it is best practice to use a switch statement because it is easier less verbose. Being easier to read, it is easier to follow the logic, and thus reduce cognitive load of reading multiple conditions.

Nevertheless, this is not a rule set in stone. It is simply a stylistic choice.

To reinforce this point, here's an example of the earlier **if else** conditional statement, using the switch syntax:

//converting the previous if-else example with switch-case

switch(light) {

   case 'green':

       console.log("Drive");

       break;

   case 'orange':

       console.log("Get ready");

       break;

   case 'red':

       console.log("Don't drive");

       break;

   default:

       //this block will run if no condition matches

       console.log('The light is not green, orange, or red');

       break;

}

# Uses of loops

In this reading, we'll discuss, at a very high level, the reasons to use loops in JavaScript.

Note that we will keep this discussion high-level because there are multiple "pieces of the puzzle" that are still missing from your understanding at this point.

This is why we will not get bogged-down in the detail of syntax and implementation, but instead, simply discuss how and why loops are used in everyday work of JavaScript developers.

Consider the following example: You work as a developer for an online store.

The store is selling letter cubes for toddlers, and the entire "Shop now" section of the site is organized in a layout where each cube on sale is displayed in a simple card component, with an image of the cube, the letter it teaches, a short description, and the price.

Cards are organized in rows, so that each row contains three cards - three different letters.

Each card is a preview of that specific letter cube on sale, and it's also a link to an entire page, dedicated to providing more info about the cubes, their teaching value, and providing the visitor with a way to complete their checkout process.

Now, here's a quick question: where would loops fit into displaying this grid of cards showcasing the letter cubes on sale?

To understand just how this works, let me code a basic prototype of how this might work.

Since you still don't have enough knowledge to display website layouts in browser with the help of JavaScript, for now I'll have to settle for using a simple string and the console.

Still, this should be a fun exercise.

var cubes = 'ABCDEFG';

//styling console output using CSS with a %c format specifier

for (var i = 0; i < cubes.length; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}

RunReset

***Note: In order to have the styles applied, try running this code snippet in your browser's console.***

That's it, with this simple code, the output in the console shows each letter on a separate line, styled like a letter cube for toddlers.

The code itself should be mostly familiar, except for the **cubes.length** and the **cubes[i]** syntax.

Without getting into too many details, here are both code snippets explained as simple as possible.

The **cubes.length** returns a number. Since **cubes** is a string of characters the **cubes.length** gives me the length of the string saved in the variable.

So this gives me the number 7, effectively making my for loop look like this:

var cubes = 'ABCDEFG';

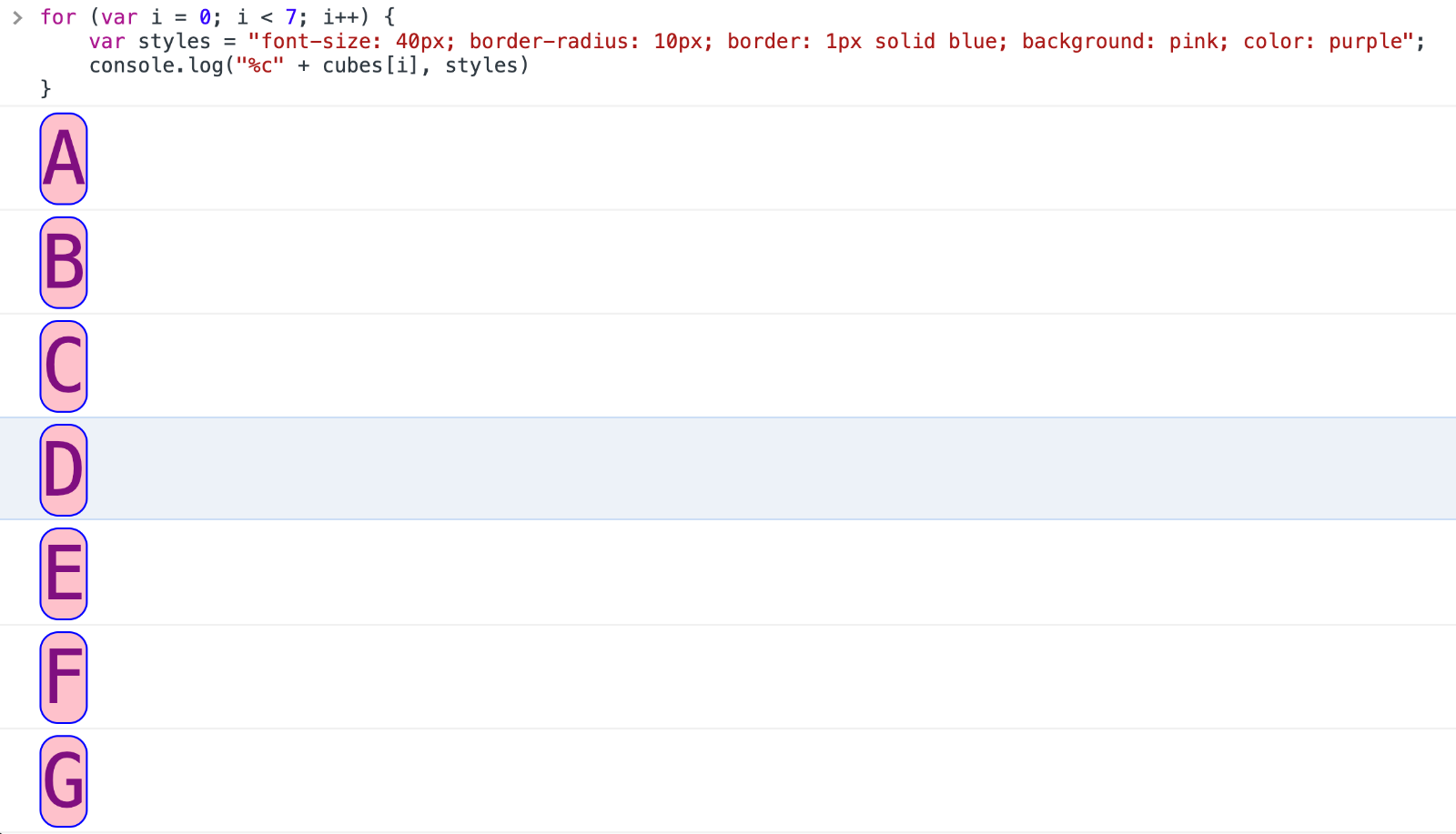
//styling console output using CSS with a %c format specifier

for (var i = 0; i < 7; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}



The second piece of code that's new here is the **cubes[i]** snippet.

This simply targets each individual letter in the loop, based on the current value of the **i** variable.

In other words, **cubes[i]**, when **i** is equal to **0**, is: **A**.

Then, **cubes[i]**, when **i** is equal to **1**, is: **B**.

This goes on for as many loops my for loop runs - and this is determined by the **cubes.length** value.

It's also very versatile, since, if I, for example, decided to change the length of the **cubes** string, I would not have to update the condition of **i < cubes.length**, because it gets automatically updated when I change the length of the **cubes** string.

There are some other ways to store data in JavaScript apps that you haven't heard about.

But we can use the same approach with those other kinds of data, to achieve results that essentially work on the same principle as the one just described.

Using loops is the essence of the approach taken in developing many different pieces of functionality in software today.

### Some additional examples

If I'm coding an email client, I will get some structured data about the emails to be displayed in the inbox, then I'll use a loop to actually display it in a nicely-formatted way.

If I'm coding an e-commerce site selling cars, I will get a source of nicely-structured data on each of the cars, then loop over that data to display it on the screen.

If I'm coding a calendar online, I'll loop over the data contained in each of the days to display a nicely-formatted calendar.

There are many, many other examples of using loops in code.

Using loops with data that is properly formatted for a given task is a crucial component of building software.

In the lessons that follow, we'll learn about different ways of grouping related data and of displaying it on the screen using JavaScript.

When combined with what you've already learned about loops, this gives you the skills to build various kinds of user interfaces where there is repetitive information.

Some more specific examples include:

* looping over blog post titles in some structured data, and displaying each blog post title on a blog home page
* looping over social media posts in some structured data, and displaying each social media post based on some conditions
* looping over some structured data on clothing available for sale in an online clothing store, and displaying relevant data for each item of clothing

Now you understand the importance of knowing how to work with loops in JavaScript. In the upcoming lessons, we'll learn other relevant information which will allow you to be able to do this.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

These resources provide some more in-depth information on the topics covered in this module.

[Mozilla Developer Network Expressions and Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators)

[Mozilla Developer Network Operator Precedence and Associativity](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

[JavaScript Primitive Values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)

[ECMA262 Specification](https://tc39.es/ecma262/)

[jQuery Official Website](https://jquery.com/)

[React Official Website](https://reactjs.org/)

[StackOverflow Developer Survey 2021 Most Popular Technologies](https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies)

[Emojis](http://unicode.org/emoji/charts/full-emoji-list.html#1f600)

# Additional resources for Conditionals and Loops

Here is a list of resources that may be helpful as you continue your learning journey.

[Comparison Operators](https://www.javascripttutorial.net/javascript-comparison-operators/)

[Truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy)

[Falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy)

[Conditional statements](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals)

In JavaScript, there is also a shorthand version of writing a conditional statement, known as the conditional (ternary) operator: [Conditional (ternary) operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)