**Intro to JavaScript**

## Meet Julia and James!

We'll be your instructors for this introductory course on JavaScript.

## JavaScript was Created for the Web

JavaScript was created in 1995 to make it easier to add interactive and dynamic elements to websites.

Today, JavaScript is used for all sorts of applications - from programming robots to writing game scripts. Even some code editors were built with JavaScript.

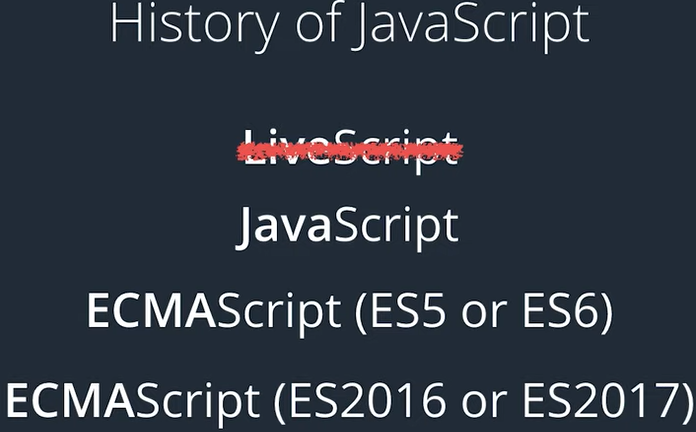
## What You Will Learn

In this course, you'll learn the foundations of the JavaScript programming language, from the history of JavaScript to how to:

* Run JavaScript code in the console of your web browser
* Create variables and use basic JavaScript data types to represent real-world data
* Use conditionals to add logic to your JavaScript programs.
* Create loops to reduce code duplication and add automation
* Write functions to streamline and organize your code.
* Use arrays to store and manipulate lists of data.
* Use objects for even more complex data organization

If you're familiar with using HTML and CSS to create web pages, JavaScript is the final piece you'll need to make your websites come to life!

# History of JavaScript



## Where Did JavaScript Come From?

The first version of JavaScript was created in just ten days 1995 by [Brendan Eich(opens in a new tab)](https://en.wikipedia.org/wiki/Brendan_Eich). Eich was working on Netscape Navigator, one of the Internet's first web browsers. Eich's goal was to add the capability for dynamic web pages, which, at that time, were simple pages of HTML and CSS.

### Why Is It Called Javascript?

JavaScript was originally called LiveScript, but it was changed back to JavaScript as a marketing decision to piggyback off Java's popularity. Despite the name, JavaScript is not related to Java in any way.

### The Evolution of JavaScript

As the language evolved, competing versions of the language emerged. To standardize the language, JavaScript was brought to [Ecma International(opens in a new tab)](https://www.ecma-international.org/), an industry association "dedicated to the standardization of information and communication systems." That's why you might hear JavaScript referred to as ***ECMAScript***.

The standards body has transitioned to a year-based number to promote a more consistent release cycle. So we have ES2016, ES2017, and so on. You can see current specifications on the ECMA website. [ECMA-262(opens in a new tab)](https://www.ecma-international.org/publications-and-standards/standards/ecma-262/), and you can even look over drafts of proposed updates [ECMAScript® 2022 Language Specification(opens in a new tab)](https://tc39.es/ecma262/)

### JavaScript Today

JavaScript has grown to be one of the most popular programming languages globally and is considered one of the foundational pillars of front-end web development.

You can read more about the current standards for JavaScript in [MDN Web Docs: JavaScript(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript)

### Want to Learn More?

Read a little history of JavaScript in [Wikipedia(opens in a new tab)](https://en.wikipedia.org/wiki/JavaScript#History).

**TIP**: HTML and CSS are markup languages\*. Markup languages are used to describe and define elements within a document. JavaScript is a\* programming language\*. Programming languages are used to communicate instructions to a machine. Programming languages can be used to control the behavior of a machine and to express algorithms.\*

## Using the JavaScript Console in Google Chrome

***Note*** You can find instructions on how to access developer tools in other browsers on the next page.

1. **Open Developer Tools**: Right-click on the page and select ***Inspect***.
2. **Open the Console**: Click on the ***Console*** tab or use a shortcut (Cmd+Option+J on macOS or Ctrl+Shift+J on Windows)
3. **Write your code!**: Here are some examples. Either type this into the browser or copy it and paste the code into the console. Don't forget to hit ***return*** to run the code.

"Julia"

**alert**("Hello, Julia! How are you?!");

Writing code in the console is a great way to test out simple bits of code, but as your code gets larger the browser user interface can be awkward. We recommend using a text editor like [VSCode(opens in a new tab)](https://code.visualstudio.com/), [Atom(opens in a new tab)](https://atom.io/), or [Sublime Text(opens in a new tab)](https://www.sublimetext.com/) to write your code and pasting your code in the console when you're ready to run it.

We'll also provide workspaces for the quizzes and activities in this course.

### Want to Learn More?

Learn about [Chrome Dev Tools Keyboard Shortcuts(opens in a new tab)](https://developers.google.com/web/tools/chrome-devtools/shortcuts) to help you move around in the console more quickly.

# Developer Tools on Different Browsers

## Developer tools on different browsers

Did you know that every modern web browser includes its own set of developer tools?

If you didn't, that's okay. Developer tools aren't always the easiest thing to find in your browser. So, we've decided to help you out by creating this guide to developer tools!

### Google Chrome

The Chrome DevTools are a set of web authoring and debugging tools built into Google Chrome. Use the DevTools to iterate, debug and profile your site. [Learn more about Chrome DevTools here(opens in a new tab)](https://developers.google.com/web/tools/chrome-devtools/).

To open Chrome DevTools, either right-click on any page element and select Inspect or open the Chrome settings menu in the top-right corner of your browser window and select More Tools > Developer Tools. Alternatively, you can use the shortcuts:

* Command + Option + i (Mac)
* Ctrl + Shift + i (Windows/Linux).

### Mozilla Firefox

Firefox Developer Tools allow you to examine, edit, and debug HTML, CSS, and JavaScript on the desktop and mobile. Also, you can download a version of Firefox called [Firefox Developer Edition(opens in a new tab)](https://www.mozilla.org/en-US/firefox/developer/), which is tailored for developers, featuring the latest Firefox features and experimental developer tools. [Learn more about Mozilla Firefox DevTools here(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Tools).

To open Firefox Developer Tools, either right-click on any page element and select Inspect Element or open the Firefox settings menu in the top-right corner of your browser window and select Developer. Alternatively, you can use the shortcuts:

* Command + Option + i (Mac)
* Ctrl + Shift + i (Windows/Linux).

### Microsoft Edge

Microsoft Edge introduced great new improvements to the F12 developer tools seen in Internet Explorer. The new tools are built in TypeScript and are always running, so no reloads are required. In addition, F12 developer tools documentation is now fully available on [GitHub(opens in a new tab)](https://github.com/MicrosoftDocs/edge-developer).

To open developer tools in Microsoft Edge, simply press F12.

[Learn more about Microsoft Edge DevTools here(opens in a new tab)](https://dev.windows.com/en-us/microsoft-edge/platform/documentation/f12-devtools-guide/).

### Safari

For any Mac users, Safari includes Web Inspector, a powerful tool that makes it easy to modify, debug, and optimize a website for peak performance and compatibility on both platforms. [Learn more about Safari Web Inspector here(opens in a new tab)](https://developer.apple.com/safari/tools/).

To access Safari's Web Development Tools, enable the Develop menu in Safari’s Advanced preferences. Once enabled, you can right-click on any page element and select Inspect Element to open Web Development Tools or use the shortcut Command + Option + i.

### Opera

Opera is a fast, lean, and powerful web browser. You can open Developer tools in Opera using the following keyboard shortcuts:

* Command + Option + i (Mac)
* Ctrl + Shift + i (Windows/Linux).

Alternatively, you can target a specific element by right-clicking on the page and selecting Inspect Element.

# console.log

## The Console Is Your Coding Sandbox

The console is a great place to mess around with your code without any long-term consequences. The console will tell you if there are any warnings or errors on the page, display any output, or print it with console.log.

### Using console.log statements

console.log is used to display content to the **JavaScript console**. Run the following code in the console:

**console**.**log**("hiya friend!");

**Prints:** "hiya friend!"

This can be very helpful in figuring out what is going on when you are debugging your code.

***NOTE***: You may see some errors or warnings in the console from the site you're visiting -- and that's okay! Warnings are very common and will not affect the code you write in this course.

### Troubleshooting

For Chrome users, if you don't see the output, click “Default levels” in the console and make sure that "Info" is checked. Congratulations! You performed the log action on the debugging console.

The message you’ve logged is "hiya friend!". hiya friend! is a **string** (a sequence of characters).

## Give It a Try!

Let’s use console.log to do something a little more interesting. Here’s a block of JavaScript code that loops through the numbers 0 through 9 and prints them out to the console:

**for** (**var** i = 0; i < 10; i++) {

**console**.**log**(i);

}

**Prints**:

0

1

2

3

4

5

6

7

8

9

This is called a **loop**.

Based on this loop's settings, any code written inside the curly brackets {...} will be repeated ten times. In this case, console.log prints out the value of i each time the loop runs. Don't worry if you're not sure about what the syntax means at this point. Later in this course, you will learn more about how and when to use loops.

# JavaScript Demo

## avaScript demo

So you saw how to use console.log to print a message to the JavaScript console. Now, let’s see how you can use the console as a sandbox to test a new line of JavaScript in the browser.

Open the [Daring Fireball website(opens in a new tab)](https://daringfireball.net/projects/markdown/) in a new tab and in that tab also open up developer tools. Then paste the following code:

document.**getElementsByTagName**("h1")[0].style.color = "#ff0000";

***NOTE***: If you can't access [*https://daringfireball.net/(opens in a new tab)*](https://daringfireball.net/), try a different website like [*https://www.udacity.com/(opens in a new tab)*](https://www.udacity.com/). This demo will work on many different websites.

Styling elements on the page is great, but you could also do that by just modifying the CSS. What makes JavaScript so special in this case? Refresh the page, then paste this line of code in the JavaScript console.

document.body.**addEventListener**('click', **function** () {

**const** myParent = document.**getElementsByTagName**("h1")[0];

**const** myImage = document.**createElement**("img");

myImage.src = 'https://thecatapi.com/api/images/get?format=src&type=gif';

myParent.**appendChild**(myImage);

myImage.style.marginLeft = "160px";

});

If you’re confused because nothing happened. Don’t worry. ***Click*** somewhere on the page to see the effect. You can refresh the page to return the page to its original state.

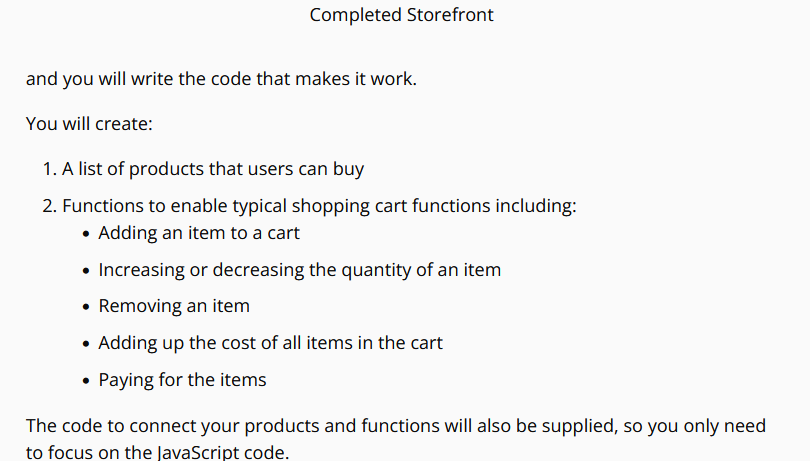
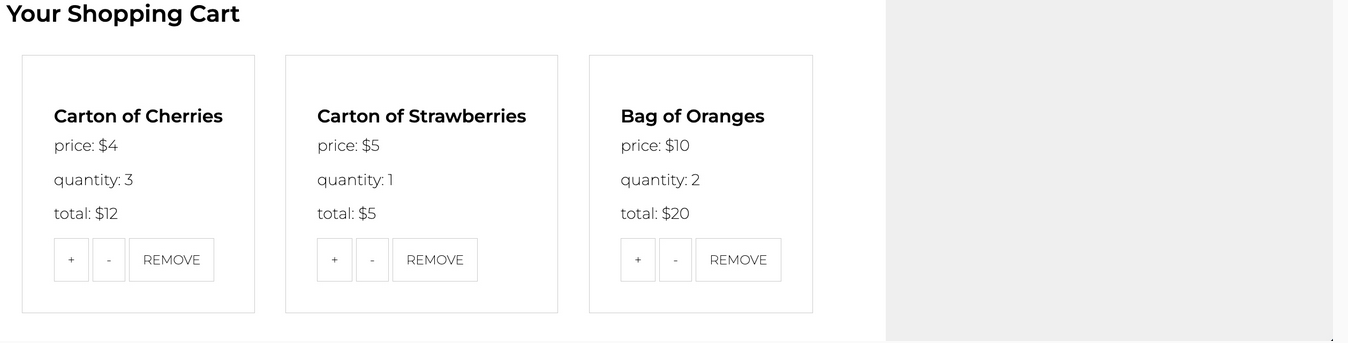
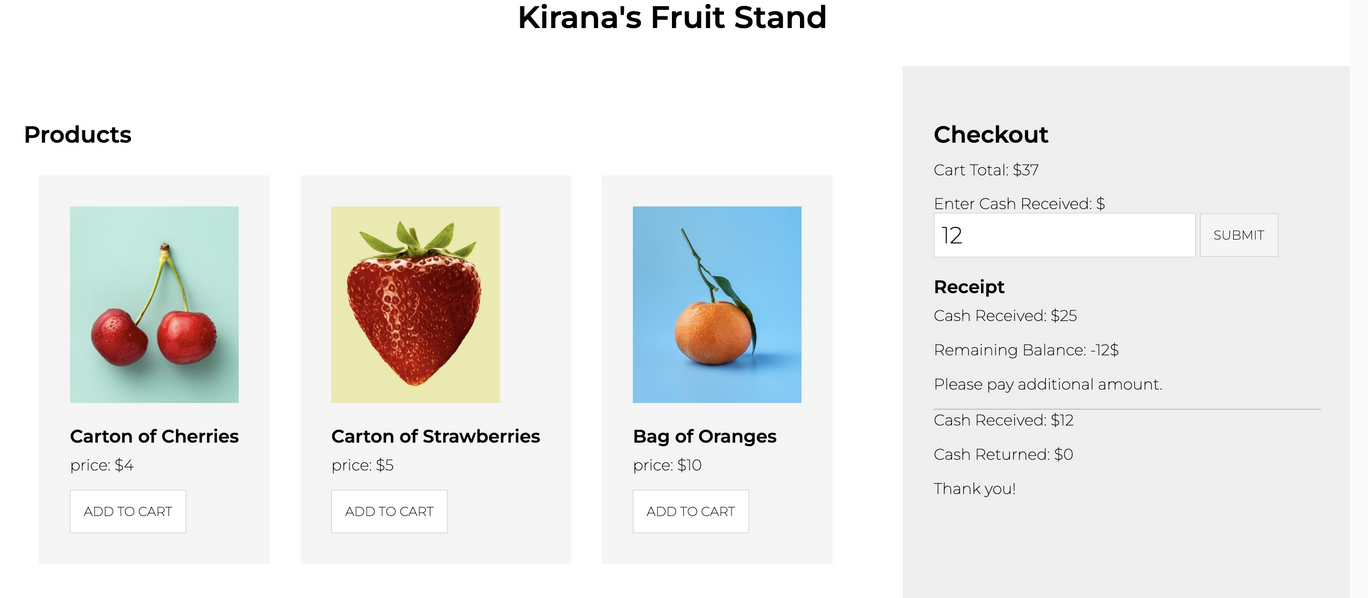
### Question 2 of 2

# Project Preview

## Build an eCommerce Shopping Cart

By the end of this course, you will understand enough JavaScript to build a fully functioning eCommerce shopping cart for Kirana's Fruit Stand.

We'll provide the front end, which looks like this:



## Why This Project?

Our goal is to challenge you to think like a programmer and use the skills you will learn in this course. This project barely touches the surface of what is possible, but it does use many of the foundational skills that will make up the majority of most of your future coding projects

# Summary

# 

## What We've Learned So Far

We've covered a lot already! You have:

* Learned that all major browsers come with built-in JavaScript engines that allow browsers to run and execute JavaScript code.
* Practiced running JavaScript code in the browser console
* Added styles to a webpage just by running code in the console

We hope you are beginning to see the power of JavaScript, and you are ready to dive in and explore the language.

# Intro to Data Types

## Data is Everywhere!

All of these things can be represented as data:

* The grade you received on your first math test.
* The conversation you had earlier today.
* The decision you made to sit down and watch this video.
* And more...

### Data is Important

It helps us:

* Understand the world
* Recognize trends
* Make educated guesses and
* Inform our decisions

Data and data types are the building blocks of any programming language because they help us organize information and determine how our programs will run.

In this lesson we will learn how to define and manipulate the primitive data types of JavaScript.

* numbers
* strings
* booleans
* undefined
* null

Once you're familiar with these data types, you'll see how you can store data in variables that you can reuse to manipulate data with your code.

# Numbers

## Numbers

Defining a number in JavaScript is actually pretty simple. The **Number** data type includes any positive or negative integer, as well as decimals. Entering a number into the console will return it right back to you.

3

**Returns:** 3

There, you did it.

## Arithmetic operations

You can also perform calculations with numbers pretty easily. Basically, type out an expression the way you would type it in a calculator.

3 + 2.1

**Returns:** 5.1

### Arithmetic Operators in JavaScript

| **Name** | **Operator** | **Meaning** |
| --- | --- | --- |
| Addition | a + b | Adds a and b |
| Subtraction | a - b | Subtracts b from a |
| Multiplication | a \* b | Multiplies a and b |
| Division | a / b | Divides a by b |
| Modulo | a % b | Returns the remainder of a / b |
| Exponent | a \*\* b | Raises a to the power of b |

#### The Modulo Operator

You are probably familiar with the basic operators, but you might not have seen the **%** operator yet.

We use the modulo operator to return the remainder of a division operation. This can be very helpful in a lot of programming tasks.

Before we move on, let's do some modulo practice.

# 

# 

# 

### Comparison Operators

| **Operator** | **Meaning** |
| --- | --- |
| < | Less than |
| > | Greater than |
| <= | Less than or Equal to |
| >= | Greater than or Equal to |
| == | Equal to |
| != | Not Equal to |

# ****TIP:**** The values true and false have significant importance in JavaScript. These values are called ****Booleans**** and are another data type in JavaScript. Later in this lesson, you’ll learn more about why Booleans are so important in programming.

# Comments

// You're about to take your first programming quiz!

Before you move onto the quiz, we want to talk about something you'll see quite often throughout this course: **comments**!

You can use **comments** to help explain your code and make things clearer. In JavaScript, comments are marked with a double forward-slash //. Anything written on the same line after the // will not be executed or displayed. To have the comment span multiple lines, mark the start of your comment with a forward-slash and star, and then enclose your comment inside a star and forward-slash / \*…\* /.

*// this is a single-line comment*

*/\**

*this is*

*a multi-line*

*comment*

*\*/*

Some of the quizzes in this course might include comments that give you hints or instructions to complete the quiz. Pay attention to those comments!

You may see comments that are used to clarify and document complex code. This may improve readability, but it often makes more sense to reduce the complexity by refactoring and simplifying the code. Simpler code is often better code.

# Quiz: First Expression

## Directions:

In the first-expressions.js file in the code editor below, write an expression that:

* uses at least 3 different arithmetic operators.
* equals 42.

**Hint:** +, -, \*, /,\* and *%* are possible arithmetic operators\*

### Running Your Code

Enter the following in the terminal:

**node** first-expression.js

[Node.js(opens in a new tab)](https://nodejs.org/en/about/) is an open-source backend JavaScript runtime environment. That basically means that it will run your JavaScript code in the terminal and return any output that is generated. We'll be using Node to run the code in our workspaces for many of the quizzes in this course.

# Connecting VS code and Node.ji run in Cmd

1. **Run Code in the Terminal**: Once installed, you can run JavaScript files by opening a terminal, navigating to the folder with your code file, and running: Node version verification

**cmd**

node -v

1. you should see the version number of Node.js displayed, meaning it’s installed correctly. Similarly, to check if NPM was installed, type:

cmd

npm -v

1. **Create folder in VS code named by “**first-expressions.js:
2. **Navigate to the Directory** in cmd where ypu saved first-expressions.js:

**Cmd**

cd Documents/….,

1. Now formulate Javascript code and save in Your VS code named by first-expressions.js:

(example)

// this expression equals 4, change it to equal 42

console.log(1 + 5 - 2);

1. Run in cmd to get result

Cmd

node first-expressions.js

# Strings: Remember “ or ‘

# 

### Key Points to Remember About Strings

* Strings can be any combination of characters -- letters, numbers and even emojis
* You must use matching quotes at the beginning or end of the string -- otherwise JavaScript thinks you are referring to a variable

**TIP:** It is correct to either use double " or single ' quotes with strings, as long as you're consistent. The [JavaScript Udacity style guide(opens in a new tab)](http://udacity.github.io/frontend-nanodegree-styleguide/javascript.html) for labs and projects suggests using single quotes to define string literals -- but your team or organization might follow a different style guide and prefer double quotes.  
**Always follow your team's style guide**

# String Concatenation

## String concatenation

**Strings** are a collection of characters enclosed inside double or single quotes. You can use strings to represent data like sentences, names, addresses, and more. Did you know you can even add strings together? In JavaScript, this is called **concatenating**. Concatenating two strings together is actually pretty simple!

"Hello," + " New York City"

**Returns:** "Hello, New York City"

You will see other ways to concatenate and do even more with strings later in this course. But for now, practice using the addition + operator.

# Variables

## Variables Allow Us to Store Data!

With variables, you no longer need to work with one-time-use data.

At the beginning of this course, you declared the value of a string, but you didn't have a way to access or reuse the string later.

"Hello"; *// Here's a String "Hello"*

"Hello" + " World"; *// Here's a new String (also with the value "Hello") concatenated with " World"*

Storing the value of a string in a variable is like packing it away for later use.

**var** greeting = "Hello";

Now, if you want to use "Hello" in a variety of sentences, you don't need to duplicate "Hello" strings. You can just reuse the greeting variable.

greeting + " World!";

**Returns:** Hello World!

greeting + " Mike!";

**Returns:** Hello Mike!

You can also change the start of the greeting by reassigning a new string value to the variable greeting.

greeting = "Hola";

greeting + " World!";

**Returns:** Hola World!

greeting + " Mike!";

**Returns:** Hola Mike!

## JavaScript Has Three Ways to Declare Variables - var, let, and const

In the example and video above, we used the keyword var to declare our variable. When we use var we create a variable that is available in the **global scope** -- which means it can be used anywhere in our program. We'll talk more about scope later, but for now, keep in mind that globally scoped variables are not a good practice.

Here's why: in a small program with just a few variables, it's easy to keep track of the variables and avoid collisions. But when you are working on a complex project or with a large team. It is very easy to inadvertently overwrite an existing variable that is hidden in another part of the program. let and const avoid this issue because they are only available in the scope where they are declared. Again, more on scope later. For now, remember:

### Best Practice #1

Always use let or const instead of var

## Which to Use - let or const?

* **Use let when we think that the value of a variable might change**  
  let ***lets*** us to assign a new value to the variable name when needed.
* **Use const when we think that the value of a variable is fixed**  
  A variable declared with const cannot be assigned a new value. It's value is ***constant***.

So which should be your default choice? That's easy. Pick the one that gives you more control: const. Using const means that your program will throw an error if there is an attempt to change the value of the variable when the code runs. If that is an intended outcome, you can go back and revise your code to declare the variable with let. If you did not intend for the value to change, congratulations! You've just discovered a bug that you can fix before it causes any problems.

### Best Practice #2

If you aren't sure, use const. You can revise your code later to replace the const with let if needed.

## Practice let and const

Try to answer these questions using what you know about let and const. If you aren't sure, open up the console and run these code snippets to see what happens.

### var still works and will remain working for the foreseeable future! 🤔

You'll see that we use var instead of let and const in some of the demo videos in this course. While all of the modern browsers support let and const, var is still part of the language. You'll see it quite often in legacy code.

## Naming conventions

When you create a variable, you should write the name of the variable using camelCase: the first word is lowercase, and all following words begin with a capital letter. Additionally, aim to use a variable name that accurately yet succinctly describes what the data represents.

*// uses camelCase if the variable name is multiple words*

**const** totalAfterTax = 53.03;

*// uses lowercase if the variable name is one word*

**const** tip = 8;

Not using camelCase for your variables names is not going to necessarily break anything in JavaScript. But there are recommended style guides used in all programming languages that help keep code consistent, clean, and easy-to-read. This is especially important when working on larger projects that will be accessed by multiple developer

# <https://google.github.io/styleguide/jsguide.html>

# 

# 

# Quiz: Converting Temperatures

To convert [Celsius(opens in a new tab)](https://en.wikipedia.org/wiki/Celsius) to [Fahrenheit(opens in a new tab)](https://en.wikipedia.org/wiki/Fahrenheit), you can use the following formula:

>F=C×1.8+32>

## Directions:

Use this equation and the variables fahrenheit and celsius to print the Fahrenheit equivalent of 12°C.

**NOTE:** "12°C" reads as "12 degrees Celsius".

# String Index

## Indexing

Did you know that you can access individual characters in a string? To access an individual character, you can use the character's location in the string, called its **index**. Just put the index of the character inside square brackets (starting with [0] as the first character) immediately after the string. For example:

"James"[0];

**Returns:** "J"

or more commonly, you will see it like this, using a variable:

**const** name = "James";

name[0];

**Returns:** "J"

# 

# 

# Escaping Strings

## Escaping strings

There are some cases where you might want to create a string that contains more than just numbers and letters. For example, what if you want to use quotes in a string?

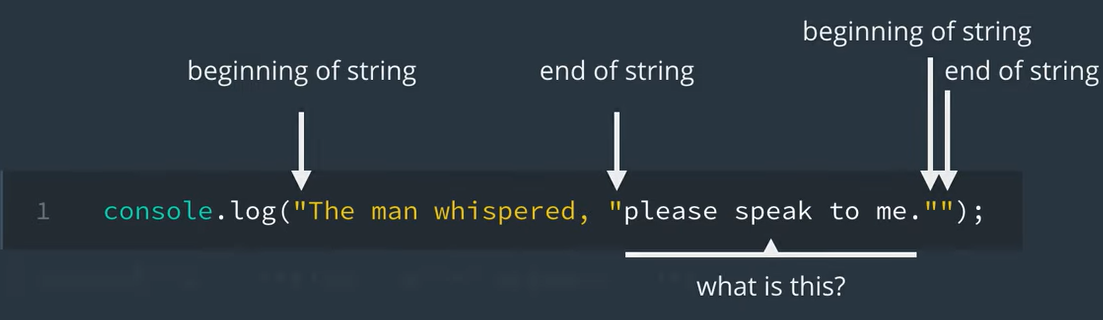
"The man whispered, "please speak to me.""

**Uncaught SyntaxError:** Unexpected identifier

If you try to use quotes within a string, you will receive a SyntaxError like the one above.

Because you need to use quotes to denote the beginning and end of strings, the JavaScript engine misinterprets the meaning of your string by thinking "The man whispered, " is the string. Then, it sees the remaining please speak to me."" and returns a SyntaxError.

If you want to use quotes inside a string, and have JavaScript not misunderstand your intentions, you’ll need a different way to write quotes. Thankfully, JavaScript has a way to do this using the backslash character ( \ ).



## Escaping characters

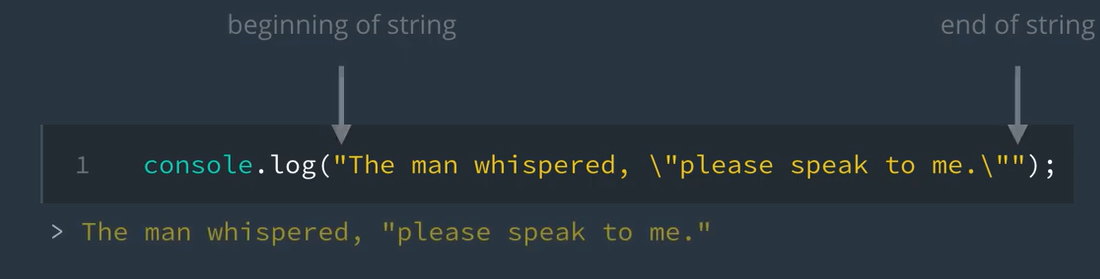
In JavaScript, you use the backslash to **escape** other characters.

Escaping a character tells JavaScript to ignore the character's special meaning and just use the literal value of the character. This is helpful for characters that have special meanings like in our previous example with quotes "…".

Because quotes are used to signify the beginning and end of a string, you can use the backslash character to escape the quotes in order to access the literal quote character.

"The man whispered, \"please speak to me.\""

**Returns:** The man whispered, "please speak to me."

This guarantees that the JavaScript engine doesn’t misinterpret the string and result in an error. 

## Special characters

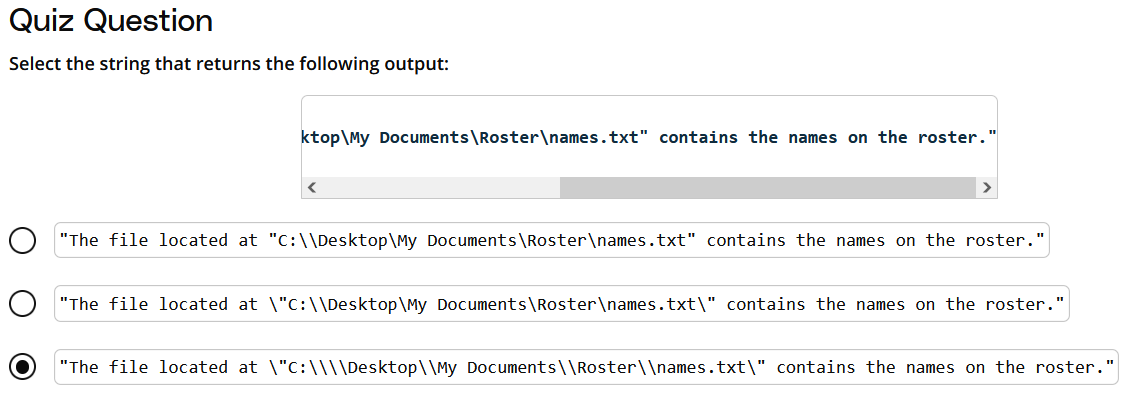
Quotes aren’t the only **special characters** that need to be escaped, there’s actually [quite a few(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Using_special_characters_in_strings). However, to keep it simple, here’s a list of some common special characters in JavaScript.

| **Code** | **Character** |
| --- | --- |
| \\ | \ (backslash) |
| " | '' (double quote) |
| ' | ' (single quote) |
| \n | newline |
| \t | tab |

The last two characters listed in the table, newline \n and tab \t, are unique because they add additional **whitespace** to your Strings. A newline character will add a line break and a tab character will advance your line to the next [tab stop(opens in a new tab)](https://en.wikipedia.org/wiki/Tab_stop).

"Up up\n\tdown down"

**Returns:**  
Up up  
 down down



## Comparing strings

Another way to work with strings is by comparing them. You've seen the comparison operators == and != when you compared numbers for equality. You can also use them with strings! For example, let’s compare the string "Yes" to "yes".

"Yes" == "yes"

**Returns:** false

When you run this in the console, it returns false. Why is that? "Yes" and "yes" are the same string, right? Well not quite.

### A. Case-sensitive

When you compare strings, case matters. While both string use the same letters (and those letters appear in the same order), the first letter in the first string is a capital Y while the first letter in the second string is a lowercase y.

'Y' != 'y'

**Returns:** true

### B. Internal Working

In Javascript, strings are compared character-by-character in alphabetical order. Each character has a specific ***numeric*** value, coming from [ASCII value of Printable characters(opens in a new tab)](https://en.wikipedia.org/wiki/ASCII#Printable_characters). For example, the character 'A' has a value 65, and 'a' has a value 97. You can notice that a lowercase letter has a higher ASCII value than the uppercase character. If you want to know the ASCII value of a particular character, you can try running the code below:

// Pick a string. Your string can have any number of characters.

const my\_string = "a";

// Calculate the ASCII value of the first character, i.e. the character at the position 0.

const ASCII\_value = my\_string.charCodeAt(0);

// Let us print

console.log(ASCII\_value);

In the example above, if you wish to print ASCII values of all the characters in your string, you would have to use ***Loops*** that we will study in later part of this course. Just for reference, here is how you can use a loop to print the ASCII value of all characters in a string.

const my\_string = "Udacity";

// Iterate using a Loop

for (let i = 0; i < my\_string.length; i++) {

console.log(my\_string.charCodeAt(i));

}

The ASCII values of [A-Z] fall in the range [65-90], whereas, the ASCII values of [a-z] fall in the range [97-122]. **Therefore, when we compare strings, the comparison happens character-by-character for the ASCII values.**

<https://www.ascii-code.com/>

# Quiz: Favorite Food

## Directions:

Create a string with the name of your favorite food. The first letter of the string should be capitalized.

### Running Your Code

Enter the following in the terminal:

**node** favorite-food.js

const favoriteFood = "Pizza";

console.log(favoriteFood);

# Quiz: String Equality for All

## Directions:

Fix the right side expression so it evaluates to true.

"ALL Strings are CrEaTeD equal" == "All STRINGS are CrEaTED Equal"

### Running Your Code

Enter the following in the terminal:

**node** equality.js

**const** answer = "ALL Strings are CrEaTeD equal" == "ALL Strings are CrEaTeD equal";

**console**.**log**(answer);

# Quiz: All Tied Up

## Directions:

Build a single string that resembles the following joke.

Why couldn't the shoes go out and play?

They were all "tied" up!

Your joke should take the format of a **question** and **answer**. The first line should be a question and the second line should be an answer.

**Hint:** You will need to use special characters to produce the following output.

### Running Your Code

Enter the following in the terminal:

**node** tied-up.js

const joke = "Why couldn't the shoes go out and play? \nThey were all \"tied\" up!"

console.**log**(joke);

# Quiz: Yosa Buson

## Directions:

Build a string using concatenation by combining the lines from this famous haiku poem by [Yosa Buson(opens in a new tab)](https://en.wikipedia.org/wiki/Yosa_Buson).

Blowing from the west

Fallen leaves gather

In the east.

Each string should be printed on its own line.

**Hint:** You will need to use special characters to produce the following output. For a refresher, feel free to review the previous ***Escaping Strings*** topic in this lesson.

### Running Your Code

Enter the following in the terminal:

**node** yosa-buson.js

**const** haiku = "Blowing from the west"+"\nFallen leaves gather"+"\nIn the east."

**console**.**log**(haiku);

# Booleans

A boolean variable can take either of two values - true or false. For example,

**const** studentName = "John";

**const** haveEnrolledInCourse = true;

**const** haveCompletedTheCourse = false;

A boolean variable is mainly essential in evaluating the outcome of conditionals (comparisons). ***The result of a comparison is always a boolean variable***. We'll study conditionals in our upcoming lesson, but let's look at our previous example to understand the role of boolean in conditional:

**if** (haveEnrolledInCourse) {

**console**.**log**("Welcome "+studentName+" to Udacity!"); *// Will run only if haveEnrolledInCourse is true*

}

Let's look at an example that will explain the role of a boolean variable in comparison.

**const** a = 10;

**const** b = 20;

*// a comparison - we will study this in detail in upcoming lesson*

**if** (a>b) { *// The outcome of a>b will be a boolean*

**console**.**log**("Variable `a` has higher value"); *// if a>b is true*

} **else** {

**console**.**log**("Variable `b` is greater than or equal to `a`"); *// if a>b is false*

}

In general cases (regular equality check), a true corresponds to number 1, whereas false represents a number 0. For example:

**if** (1) {

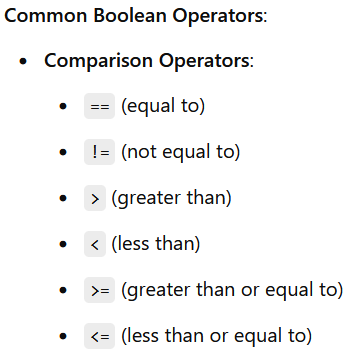
**console**.**log**("This statement will always execute because conditional is set to 1 i.e., true");

}

**if** (0) {

**console**.**log**("This statement will NEVER execute because conditional is set to 0 i.e., false");

}



# Quiz: Facebook Post

### Quiz Question

Look at the picture below and identify which data type (Numbers, Strings, Booleans) would best represent the highlighted areas.

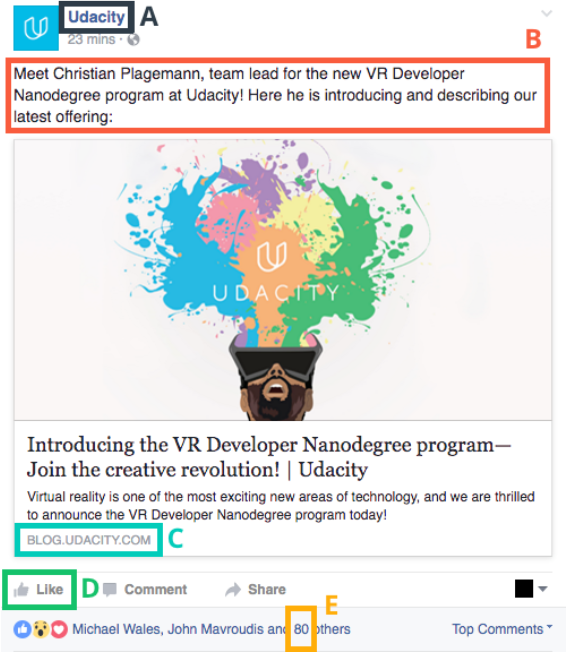
**Hint:** They’re not all Strings

Here's the analysis of the highlighted areas for data types:

1. **A (Udacity Logo):** This would likely be a **String** data type because it represents a brand name ("Udacity").
2. **B (Text introducing Christian Plagemann):** This is a descriptive text, so it would also be a **String**.
3. **C (URL):** This is a web address and would be represented as a **String** (even though it might contain numbers, URLs are generally stored as strings).
4. **D (Like/Comment/Share buttons):** These can be **Booleans** as each represents an interaction state (clicked or not clicked).
5. **E (Number of likes "80"):** This is a **Number** because it represents a count.

To summarize:

* **A:** String
* **B:** String
* **C:** String
* **D:** Booleans
* **E:** Number



# Null, Undefined, and NaN

### What is the Difference Between null and undefined?

null refers to the "value of nothing", while undefined refers to the "absence of value".

## What is NaN?

NaN stands for "Not-A-Number" and it's often returned indicating an error with number operations. For instance, if you wrote some code that performed a math calculation, and the calculation failed to produce a valid number, NaN might be returned.

*// calculating the square root of a negative number will return NaN*

**Math**.**sqrt**(-10)

*// trying to divide a string by 5 will return NaN*

"hello"/5

### 

# Equality

## Equality

So far, you’ve seen how you can use == and != to compare numbers and strings for equality. However, if you use == and != in situations where the values that you're comparing have different data-types, it can lead to some interesting results. For example,

"1" == 1

**Returns:** true

and

0 == false

**Returns:** true. The == operator is unable to differentiate 0 from false.

' ' == false

**Returns:** true. Both the operands on either side of the == operator are first converted to zero, before comparison.

All of the above three evaluate to true. The reason for such interesting outcomes is ***Type Conversion***. In the case of regular comparison, the operands on either side of the == operator are first converted to numbers, before comparison. Therefore, a ' ', false, and 0 are all considered equal. Similarly, a '1' and 1 are also considered equal. If we don't want to convert the operands, before comparison, we have to use a **strict comparison** ===, that is explained below.

## Implicit Type Coercion

JavaScript is known as a loosely typed language.

Basically, this means that when you’re writing JavaScript code, you do not need to specify data types. Instead, when your code is interpreted by the JavaScript engine it will automatically be converted into the "appropriate" data type. This is called implicit type coercion and you’ve already seen examples like this before when you tried to concatenate strings with numbers.

"julia" + 1

**Returns:** "julia1"

In this example, JavaScript takes the string "julia" and adds the number 1 to it resulting in the string "julia1". In other programming languages, this code probably would have returned an error, but in JavaScript the number 1 is converted into the string "1" and then is concatenated to the string "julia".

It’s behavior like this which makes JavaScript unique from other programming languages, but it can lead to some quirky behavior when doing operations and comparisons on mixed data types.

## Strongly Typed vs Loosely Typed

A **strongly typed language** is a programming language that is more likely to generate errors if data does not closely match an expected type.

**Example of strongly typed programming language code**

int count = 1;

string name = "Julia";

double num = 1.2932;

float price = 2.99;

With a **loosely typed language** like JavaScript, you don’t need to specify data types; this provides a lot more flexibility and is often faster to write. However, loose typing can lead to errors that are hard to diagnose due to implicit type coercion.

**Equivalent code in JavaScript**

*// equivalent code in JavaScript*

**let** count = 1;

**const** name = "Julia";

**const** num = 1.2932;

**const** price = 2.99;

## Strict vs. Loose Equality

In the example below, JavaScript takes the string "1", converts it to true, and compares it to the boolean true.

"1" == true

**Returns:** true

When you use the == or != operators, JavaScript first converts each value to the same type (if they’re not already the same type); this is why it's called "type coercion"! This is often not the behavior you want, and **it’s actually considered bad practice to use the == and != operators when comparing values for equality**.

Instead, in JavaScript it’s better to use **strict equality** to see if numbers, strings, or booleans, etc. are identical in type and value without doing the type conversion first. To perform a strict comparison, simply add an additional equals sign = to the end of the == and != operators.

"1" === 1

**Returns:** false

This returns false because the string "1" is not the same type and value as the number 1.

0 === false

**Returns:** false

This returns false because the number 0 is not the same type and value as the boolean false. Just like strict equality operator, there is also a ***strict non-equality*** operator !== that can be used instead of != if you don't want a type-conversion, before comparison. For example,

0 !== true

**Returns:** true

and

'1' !== 1

**Returns:** true

In JavaScript, when comparing values, you have two main options:

Loose Equality (== and !=): This comparison allows JavaScript to automatically convert different data types (like numbers and strings) to a common type before comparing them. This is called type coercion.

Strict Equality (=== and !==): This comparison checks both the value and the data type without any conversion.

Loose Equality: == and !=

Using == or != lets JavaScript convert (or "coerce") one data type into another before making the comparison. This can lead to some unexpected results:

"1" == 1

Here, the string "1" is converted to the number 1, so the comparison is actually 1 == 1, which is true.

0 == false

JavaScript converts false to 0, making the comparison 0 == 0, which is true.

' ' (a single space) == false

Both sides are coerced to 0 because JavaScript considers an empty string or a single space as "falsy," so it becomes 0 == 0, which is true.

Loose equality is flexible but can make code unpredictable because different data types might compare as equal.

Strict Equality: === and !==

Strict equality solves this by checking both value and data type without converting anything. Only if both are identical will it return true.

"1" === 1

The types are different: "1" is a string, and 1 is a number. No conversion happens, so this comparison returns false.

0 === false

0 is a number, and false is a boolean, so without type conversion, they are different types, and the comparison returns false.

0 !== true

Here, 0 (a number) is not the same type or value as true (a boolean), so the comparison returns true.

Key Takeaways

Loose equality (==) allows JavaScript to convert types, which can sometimes create confusing results.

Strict equality (===) checks type and value without converting, giving clearer, more predictable outcomes.

Whenever possible, using strict equality (=== and !==) in JavaScript is recommended to avoid unexpected behavior due to type conversion.

const strValue = "3";

const numValue = 1;

const result = strValue > numValue;

console.log(`Is "${strValue}" greater than ${numValue}?`, result);

const numValue = 3;

const strValue = "3";

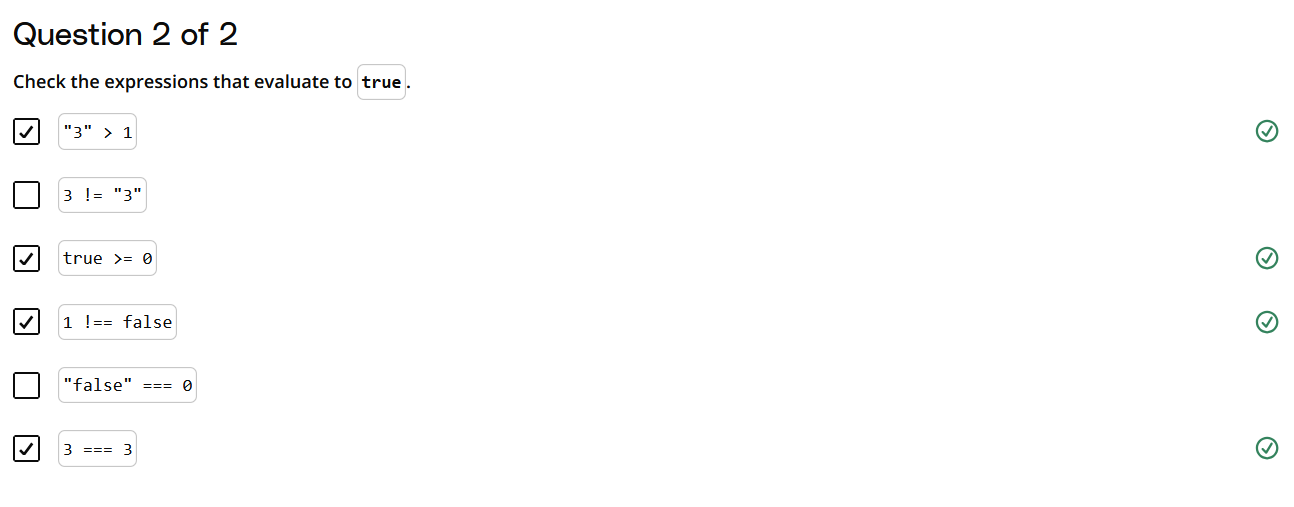
const result = numValue != strValue;

console.log(`Is ${numValue} not equal to "${strValue}"?`, result);

const boolValue = true;

const result = boolValue >= 0;

console.log(`Is true greater than or equal to 0?`, result);



# Quiz: Semicolons!

One thing to take notice of in all the examples you've seen so far is the use of semicolons ; at the end of each line. Semicolons make it clear where one statement ends and another begins. This is especially handy when multiple lines of code are written on the same line (which is valid JavaScript, but definitely not recommended!). For instance:

**const** totalAfterTax = 53.03 **let** tip = 8 *// this is incorrect!*

**Uncaught SyntaxError:** Unexpected token var

vs.

**const** totalAfterTax = 53.03; **let** tip = 8; *// this is correct!*

Not adding semicolons to the end of each line can cause bugs and errors in your programs. JavaScript does have ways to occasionally predict where semicolons should be, but just like how type coercion can result in some unexpected quirky behavior in JavaScript, it's good practice to not depend on it.

## Directions for the Quiz Below:

Define two variables called thingOne and thingTwo and assign them values. Print the values of both variables in **one** console.log statement using **concatenation**. For example,

red blue

where "red" is the value of thingOne and "blue" is the value of thingTwo. **Don't forget to use semicolon at the end of each statement**!

### Running Your Code

Enter the following in the terminal:

**const** thingOne = "Text One";

**const** thingTwo = "Text Two";

**console**.**log**(thingOne + " " + thingTwo);

# Quiz: What's my Name?

## Directions:

Create a variable called fullName and assign it your full name as a string. Print the value of the variable in the console.

### Running Your Code

Enter the following in the terminal:

const fullName = "Grace Hopper";

console.log(fullName);

# Quiz: Out to Dinner

## Directions:

Create a variable called bill and assign it the result of 10.25 + 3.99 + 7.15 (don't perform the calculation yourself, let JavaScript do it!). Next, create a variable called tip and assign it the result of multiplying bill by a 15% tip rate. Finally, add the bill and tip together and store it into a variable called total.

Print the total to the JavaScript console.

**Hint:** 15% in decimal form is written as 0.15.

**TIP:** To print out the total with a dollar sign ( $ ) use string concatenation. To round total up by two decimal points use the toFixed() method. To use toFixed() pass it the number of decimal points you want to use. For example, if total equals 3.9860, then total.toFixed(2) would return 3.99.

### Running Your Code

Enter the following in the terminal:

const bill = 10.25 + 3.99 + 7.15;

const tip = 0.15 \* bill;

const total = bill + tip;

console.log("$"+total);

# Quiz: Mad Libs

## Directions:

[Mad Libs(opens in a new tab)](https://en.wikipedia.org/wiki/Mad_Libs) is a word game where players have fun substituting words for blanks in a story. For this exercise, use the adjective variables below to fill in the blanks and complete the following message.

"The Intro to JavaScript course is \_\_\_\_\_\_\_\_\_\_. James and Julia are

so \_\_\_\_\_\_\_\_\_\_. I cannot wait to work through the rest of

this \_\_\_\_\_\_\_\_\_\_ content!"

**const** adjective1 = "amazing";

**const** adjective2 = "fun";

**const** adjective3 = "entertaining";

Assign the resulting string to a variable called madLib.

### Running Your Code

Enter the following in the terminal:

const adjective1 = "amazing";

const adjective2 = "fun";

const adjective3 = "entertaining";

const madLib = "The Intro to JavaScript course is "+ adjective1 +". James and Julia are so "+ adjective2 +". I cannot wait to work through the rest of this "+ adjective3 +" content!";

console.log(madLib);

# Quiz: One Awesome Message

## Directions:

Here are two awesome messages:

Hi, my name is Julia. I love cats. In my spare time, I like to play video games.

Hi, my name is James. I love baseball. In my spare time, I like to read.

Declare and assign values to three variables for each part of the sentence that changes (firstName, interest, and hobby).

Use your variables and string concatenation to create your own awesome message and store it in an awesomeMessage variable. Finally, print your awesome message to the JavaScript console.

### Running Your Code

Enter the following in the terminal:

const firstName = "Julia";

const interest = "cats";

const hobby = "play video games";

const awesomeMessage = "Hi, my name is " + firstName + ". I love " + interest + ". In my spare time, I like to " + hobby + ".";

console.log(awesomeMessage);

# Intro to Conditionals

## We Write Code to Solve Problems

When you write code, you break down problems into steps that are executed by a computer. These steps are known as an **algorithm**.

***Example: Purchase Algorithm***

Think about the steps you need to take to decide whether to purchase an item.

1. Do I want the item?
2. Do I have enough money to purchase the item?

If the answer to both questions is yes, you will purchase the item. The process you went through to solve that problem, and the steps you took in order to do so, is at the heart of writing code.

In this lesson we'll learn about the **conditional** statements we can use in JavaScript to execute the algorithms we need to solve problems.

### What We Will Cover in This Lesson

You will learn how to:

* Write if...else and else...if conditionals.
* Use logical operators to handle more complex logic.
* Identify truthy and falsy values.
* Use ternary operators for more concise conditional logic.
* Use a switch statement to chain multiple else if statements.

Let's get started!

# Quiz: Flowcharts

# **DEFINITION:** A **flowchart** is a visual diagram that outlines the solution to a problem through a series of logical statements. The order in which statements are evaluated and executed is called the **control flow**.

# 

# 

# Flowchart to Code

## Solving the Problem: Do I purchase the hammer?

To answer that question we need to know:

* How much money do I have?
* How much does the hammer cost?

Let's translate that into code:

**const** price = 15.00;

**const** money = 20.00;

Now we compare the price to the money we have to determine ***if*** we have enough money. We can use a JavaScript if else statement for that:

**if** (money >= price) {

**console**.**log**("buy the hammer");

} **else** {

**console**.**log**("don't buy the hammer");

}

Pay attention to the curly braces! The code in the curly braces after the if statement will run ***if*** the statement evaluates to true and the code in the curly braces after the else statement will run if it evaluates to false.

### Why does the video use var and the example code use const?

Although var works, it is now considered best practice to use let or const instead. We're showing you both but you should try to use let and const in your own code.

## If...else Statements

**If...else statements** allow you to execute certain pieces of code based on a condition, or set of conditions, being met.

**if** (/ \*this expression is true\* /) {

*// run this code*

} **else** {

*// run this code*

}

This is extremely helpful because it allows you to choose which piece of code you want to run based on the result of an expression. For example,

**const** a = 1;

**const** b = 2;

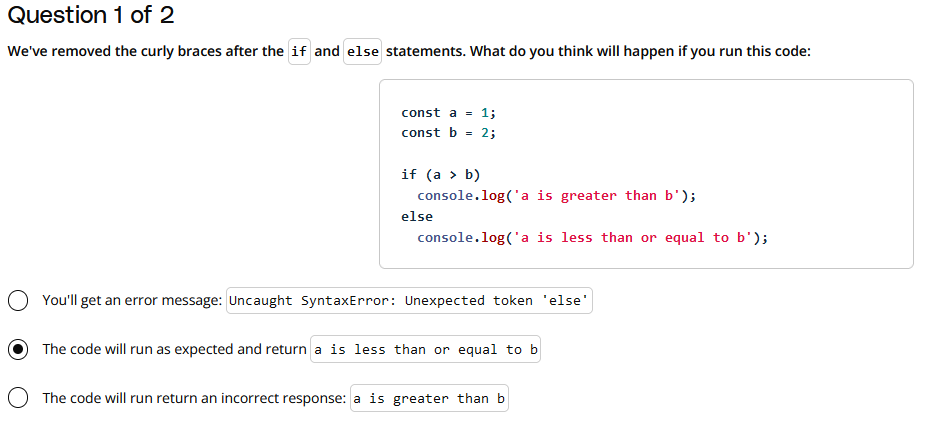
**if** (a > b) {

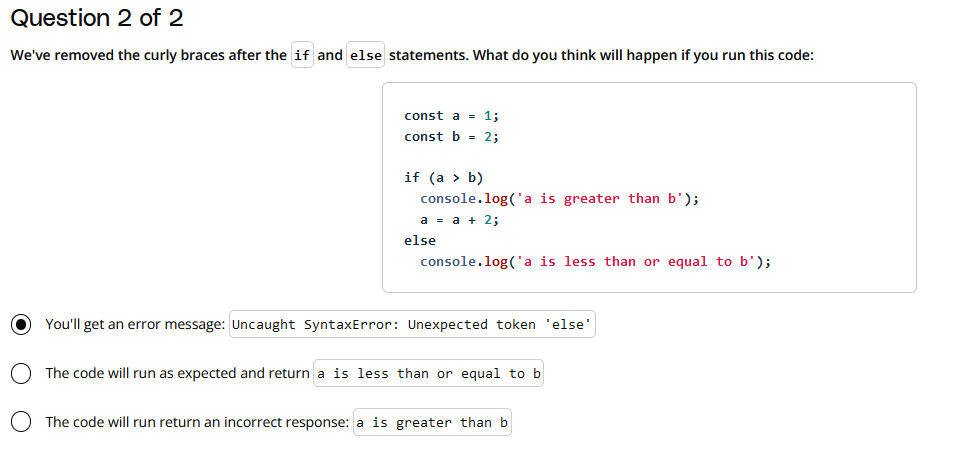
**console**.**log**("a is greater than b");

} **else** {

**console**.**log**("a is less than or equal to b");

}





#### What We Learned

Curly braces are not necessary if you have only one line of code to execute following an if or else statement. They are necessary if you have more than one line of code to execute.

That being. said, in most cases, even though it is not required, **it is a better practice to use curly braces whenever you are using a conditional statement.**

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else>

# Else If Statements

By adding the extra else if statement, you're adding an **extra conditional statement.**

If it’s not going to snow, then the code will jump to the else if statement to see if it’s going to rain. If it’s not going to rain, then the code will jump to the else statement.

The else statement essentially acts as the "default" condition in case all the other if statements are false.

const weather = "sunny";

if (weather === "snow") {

  console.log("Bring a coat.");

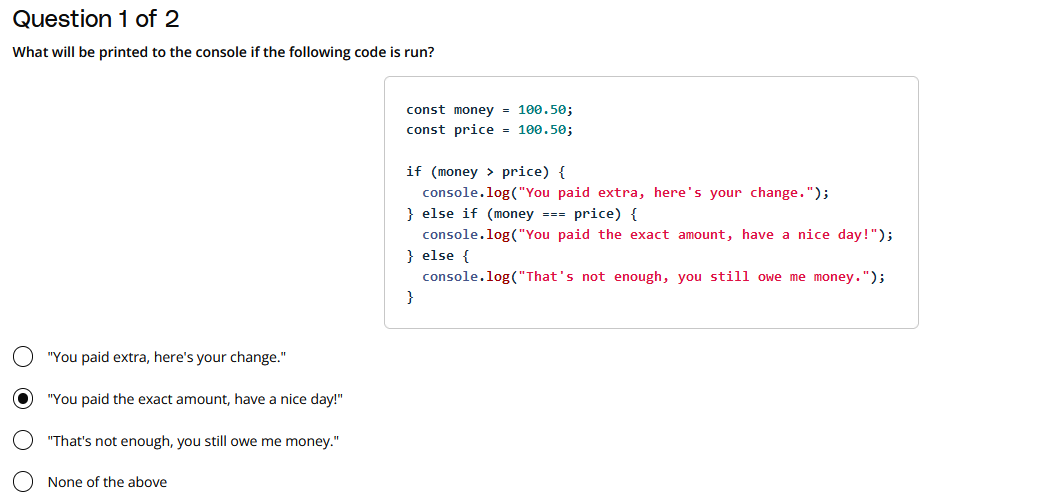
} else if (weather === "rain") {

  console.log("Bring a rain jacket.");

} else {

  console.log("Wear what you have on.");

}



<https://video.udacity-data.com/topher/2017/January/586e9836_what-to-wear-flowchart/what-to-wear-flowchart.jpeg>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else#description>

# Quiz: Even or Odd

## Directions:

Write an if...else statement that:

* prints "even" if the number is an even number
* prints "odd" if the number is an odd number

**Hint:** Use the % ([modulo(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Remainder_())) operator to determine if a number is even or odd. The modulo operator takes two numbers and returns the remainder when the first number is divided by the second one:

**console**.**log**(12 % 3);

**console**.**log**(10 % 4);

**Result**:

0

2

The answer for 12 % 3 is 0 because twelve divided by three has no remainder. 10 % 4 is 2 because ten divided by 4 has a remainder of two.

Make sure to test your code with different values. For example:

* If number equals 1, then odd should be printed to the console.
* If number equals 12, then even should be printed to the console.

### Running Your Code

Enter the following in the terminal:

**node** even-or-odd.js

[Node.js(opens in a new tab)](https://nodejs.org/en/about/) is an open-source backend JavaScript runtime environment. That basically means that it will run your JavaScript code in the terminal and return any output that is generated. We'll be using Node to run the code in our workspaces for many of the quizzes in this course.

const number = 12;

if (number % 2 === 0) {

  console.log("even");

} else {

  console.log("odd");

}

*cd "C:\Users\Hayyuu\OneDrive - UGent\Desktop\Programming Fundamentals\Javapractice"*

*nod xxxx.js*

# Quiz: Musical Groups

Musical groups have special names based on the number of people in the group.

For example, a "quartet" is a musical group with four musicians. [Barbershop quartets(opens in a new tab)](https://en.wikipedia.org/wiki/Barbershop_quartet) were a popular type of quartet in the early 1900s and featured four singers made up of a lead, tenor, baritone, and bass.

## Directions:

Write a series of conditional statements that:

* Prints "not a group" if musicians is less than or equal to 0
* Prints "solo" if musicians is equal to 1
* Prints "duet" if musicians is equal to 2
* Prints "trio" if musicians is equal to 3
* Prints "quartet" if musicians is equal to 4
* Prints "this is a large group" if musicians is greater than 4

***HINT*** If you aren't sure how to proceed, try drawing a flow chart to outline the decision points and how the decision is made at each point

### Running Your Code

Enter the following in the terminal:

**node** musical-groups.js

### Testing Your Code

How will you know if your code works? Change the value of number and re-run the code.

| **Value of musicians** | **Output** |
| --- | --- |
| 0 | "not a group" |
| 1 | "solo" |
| 2 | "duet" |
| 3 | "trio" |
| 4 | "quartet" |
| 76 | "this is a large group" |

// change the value of ,[object Object], to test your conditional statements

const musicians = 0;

// your code goes here

if (musicians <= 0) {

    console.log("not a group");

    } else if (musicians === 1) {

    console.log("solo");

    } else if (musicians === 2) {

    console.log("duet");

    } else if (musicians === 3) {

    console.log("trio");

    } else if (musicians === 4) {

    console.log("quartet");

    } else if (musicians >= 5) {

    console.log("this is a large group");

    }

# More Complex Problems

## Problems Aren't Always Simple

With most problems there's more things to consider before you can actually solve the problem.

***Example: Deciding what to do this weekend***

Julia's plan is to hang out with her friend Colt at the park.

For Julia's plans to work:

1. Colt must be available
2. The weather must be nice

In JavaScript we can represent complex problems by combining logical expressions with special operators called ***logical operators***.

# Logical Operators

Here’s the logical expression used to represent Julia’s weekend plans:

**var** colt = "not busy";

**var** weather = "nice";

**if** (colt === "not busy" && weather === "nice") {

**console**.**log**("go to the park");

}

**Prints:** "go to the park"

Notice the && in the code above.

The && symbol is the logical AND operator, and it is used to combine two logical expressions into one larger logical expression. If **both** smaller expressions are true, then the entire expression evaluates to true. If **either one** of the smaller expressions is false, then the whole logical expression is false.

Another way to think about it is when the && operator is placed between the two statements, the code literally reads, "if Colt is not busy AND the weather is nice, then go to the park".

## Logical expressions

**Logical expressions** are similar to mathematical expressions, except logical expressions evaluate to either true or false.

11 != 12

**Returns:** true

You’ve already seen logical expressions when you write comparisons. A comparison is just a simple logical expression.

Similar to mathematical expressions that use +, -, \*, / and %, there are logical operators &&, || and ! that you can use to create more complex logical expressions.

## Logical operators

**Logical operators** can be used in conjunction with boolean values (true and false) to create complex logical expressions.

By combining two boolean values together with a logical operator, you create a logical expression that returns another boolean value. Here’s a table describing the different logical operators:

| **Operator** | **Meaning** | **Example** | **How it works** |
| --- | --- | --- | --- |
| && | Logical AND | value1 && value2 | Returns true if **both** value1 **and** value2 evaluate to true. |
| || | Logical OR | value1 || value2 | Returns true if **either** value1 **or** value2 (**or even both!**) evaluates to true. |
| ! | Logical NOT | !value1 | Returns the **opposite** of value1. If value1 is true, then !value1 is false. |

By using logical operators, you can create more complex conditionals like Julia’s weekend example.

# *****TIP:***** *Logical expressions are evaluated from left to right. Similar to mathematical expressions, logical expressions can also use parentheses to signify parts of the expression that should be evaluated first.*

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_operators#logical_operators>

# Logical AND and OR

# 

# Quiz: Murder Mystery

## Directions:

For this quiz, you're going to help solve a fictitious [murder mystery(opens in a new tab)](https://en.wikipedia.org/wiki/Murder_mystery_game) that happened here at Udacity! A murder mystery is a game typically played at parties wherein one of the partygoers is secretly, and unknowingly, playing a murderer, and the other attendees must determine who among them is the criminal. It's a classic case of [whodunnit(opens in a new tab)](https://en.wikipedia.org/wiki/Whodunit).

Since this might be your first time playing a murder mystery, we've simplified things quite a bit to make it easier. Here's what we know! In this murder mystery there are:

* **four rooms**: the ballroom, gallery, billiards room, and dining room,
* **four weapons**: poison, a trophy, a pool stick, and a knife,
* and **four suspects**: Mr. Parkes, Ms. Van Cleve, Mrs. Sparr, and Mr. Kalehoff.

We also know that each weapon corresponds to a particular room, so...

* the poison belongs to the ballroom,
* the trophy belongs to the gallery,
* the pool stick belongs to the billiards room,
* and the knife belongs to the dining room.

And we know that each suspect was located in a specific room at the time of the murder.

* Mr. Parkes was located in the dining room.
* Ms. Van Cleve was located in the gallery.
* Mrs. Sparr was located in the billiards room.
* Mr. Kalehoff was located in the ballroom.

To help solve this mystery, write a combination of conditional statements that:

1. sets the value of weapon based on the room and
2. sets the value of solved to true if the value of room matches the suspect's room

Afterwards, use this template to print a message to the console if the mystery was solved:

\_\_\_\_\_\_\_\_\_\_ did it in the \_\_\_\_\_\_\_\_\_\_ with the \_\_\_\_\_\_\_\_\_\_!

What goes into the three blank spaces? You can fill in the blanks with the name of the suspect, the room, and the weapon! For example, an output string may look like:

Mr. Parkes did it in the dining room with the knife!

let room = "dining room"; // Change this value to test different rooms

let suspect = "Mr. Parkes"; // Change this value to test different suspects

let weapon;

let solved = false;

if (room === "ballroom") {  // Determine the weapon based on the room

    weapon = "poison";

} else if (room === "gallery") {

    weapon = "trophy";

} else if (room === "billiards room") {

    weapon = "pool stick";

} else if (room === "dining room") {

    weapon = "knife";

}

if (

    (suspect === "Mr. Parkes" && room === "dining room") || // Check if the suspect is in the correct room

    (suspect === "Ms. Van Cleve" && room === "gallery") ||

    (suspect === "Mrs. Sparr" && room === "billiards room") ||

    (suspect === "Mr. Kalehoff" && room === "ballroom")

) {

    solved = true; // Print the result based on whether the case is solved or not

}

if (solved) {

    console.log(`${suspect} did it in the ${room} with the ${weapon}!`);

} else {

    console.log("The case is not solved!");

}

### Expected Outcomes

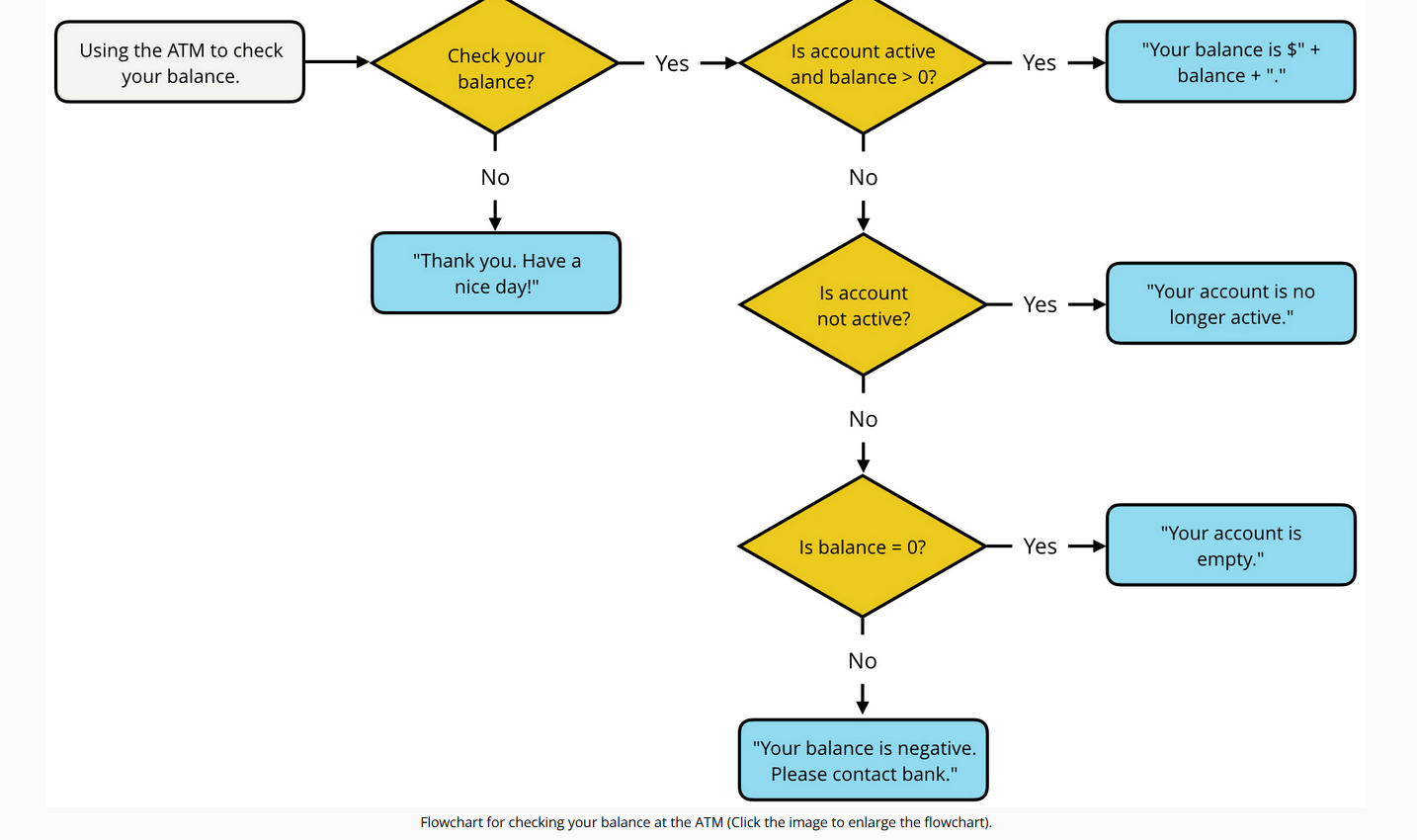
* You can test the following scenarios based on the expected outcomes:

1. room = "dining room"; suspect = "Mr. Parkes"; → Output: **"Mr. Parkes did it in the dining room with the knife!"**
2. room = "gallery"; suspect = "Ms. Van Cleve"; → Output: **"Ms. Van Cleve did it in the gallery with the trophy!"**
3. room = "billiards room"; suspect = "Mrs. Sparr"; → Output: **"Mrs. Sparr did it in the billiards room with the pool stick!"**
4. room = "ballroom"; suspect = "Mr. Kalehoff"; → Output: **"Mr. Kalehoff did it in the ballroom with the poison!"**
5. room = "billiards room"; suspect = "Ms. Van Cleve"; → Output: **"The case is not solved!"**
6. room = "dining room"; suspect = "Mrs. Sparr"; → Output: **"The case is not solved!"**
7. room = "gallery"; suspect = "Mr. Kalehoff"; → Output: **"The case is not solved!"**
8. room = "ballroom"; suspect = "Mr. Parkes"; → Output: **"The case is not solved!"**

# Quiz: Checking your Balance

## Directions:

Using the flowchart below, write the code to represent checking your balance at the ATM. The yellow diamonds represent conditional statements and the blue rectangles with rounded corners represent what should be printed to the console.



Use the following variables in your solution:

* balance - the account balance
* isActive - if account is active
* checkBalance - if you want to check balance

**Hint:** The variable balance could be a value less than, greater than, or equal to 0. The variables isActive and checkBalance are booleans that can be set to true or false.

**TIP:** To print out the account balance with decimal points (i.e. 325.00), use the .toFixed() method and pass it the number of decimal points you want to use. For example, balance.toFixed(2) returns 325.00.

### Running Your Code

Enter the following in the terminal:

**node** balance.js

### Testing Your Code

How will you know if your code works? Change the values of balance, checkBalance and isActive and re-run the code and compare your output with the expected output.

#### Expected Output

| **Customer** | **Balance** | **checkBalance** | **isActive** | **Expected Output** |
| --- | --- | --- | --- | --- |
| Customer1 | -325 | true | true | Your balance is negative. Please contact bank. |
| Customer2 | 35 | true | true | Your balance is $35.00. |
| Customer3 | 35 | false | true | Thank you. Have a nice day! |
| Customer4 | 35 | true | false | Your account is no longer active. |
| Customer5 | 0 | true | true | Your account is empty. |
| Customer6 | -325 | false | true | Thank you. Have a nice day! |
| Customer7 | -325 | true | false | Your account is no longer active. |
| Customer8 | 35 | false | false | Thank you. Have a nice day! |
| Customer9 | 0 | false | false | Thank you. Have a nice day! |
| Customer10 | 0 | true | false | Your account is no longer active. |

// Define variables

let balance = 0;       // Example value, can be changed for testing

let isActive = true;   // Example value, can be changed for testing

let checkBalance = true; // Example value, can be changed for testing

if (checkBalance) {  // Implementing the logic from the flowchart

    if (isActive && balance > 0) {

        console.log(`Your balance is $${balance.toFixed(2)}.`);

    } else if (!isActive) {

        console.log("Your account is no longer active.");

    } else if (balance === 0) {

        console.log("Your account is empty.");

    } else {

        console.log("Your balance is negative. Please contact bank.");

    }

} else {

    console.log("Thank you. Have a nice day!");

}

### Testing with Different Scenarios

You can change the values of balance, isActive, and checkBalance to test each scenario as shown in the "Expected Output" table. This will confirm if the code matches the expected behavior from the flowchart.

# Quiz: Ice Cream

## Directions:

Ice cream is one of the most versatile desserts on the planet because it can be done up so many different ways and every ice cream shop offers a different set of flavors, toppings and serving vessels.

Using logical operators, write a series of complex logical expressions that evaluates a cusomer order and prints an approriate response.

If the order can be filled, respond with:

"Great choice! Your ice cream is at the next window."

If the order cannot be filled, respond with:

"Please check our menu and try again."

Our Ice Cream Store offers the following:

* Flavors: vanilla or chocolate
* Topping: sprinkles or peanuts
* Vessel: wafer cone or sugar cone

### Running Your Code

Enter the following in the terminal:

**node** ice-cream.js

### Testing Your Code

How will you know if your code works? Change the values of flavor, topping and vessel and re-run the code and compare your output with the expected output.

#### Expected Output

| **Flavor** | **Topping** | **Vessel** | **Response** |
| --- | --- | --- | --- |
| chocolate | bananas | wafer cone | "Please check our menu and try again." |
| chocolate | peanuts | wafer cone | "Great choice! Your ice cream is at the next window." |
| chocolate | sprinkles | sugar cone | "Great choice! Your ice cream is at the next window." |
| chocolate | sprinkles | bowl | "Please check our menu and try again." |
| strawberry | sprinkles | wafer cone | "Please check our menu and try again." |
| strawberry | bananas | sugar cone | "Please check our menu and try again." |
| strawberry | peanuts | bowl | "Please check our menu and try again." |
| vanilla | sprinkles | wafer cone | "Great choice! Your ice cream is at the next window." |
| vanilla | peanuts | sugar cone | "Great choice! Your ice cream is at the next window." |
| vanilla | sprinkles | bowl | "Please check our menu and try again." |

const flavor = "strawberry";

const topping = "sprinkles";

const vessel = "wafer cone";

// your code here

if ((flavor === "vanilla" || flavor === "chocolate") &amp;&amp; (topping === "sprinkles" || topping === "peanuts") &amp;&amp;(vessel === "sugar cone" || vessel === "wafer cone")){

    console.log ("Great choice! Your ice cream is at the next window.");

} else {

    console.log ("Please check our menu and try again.");

}

# Quiz: What do I Wear?

If you're like me, finding the right size t-shirt can sometimes be a challenge. What size am I? What's the difference between S (small), M (medium), and L (large)? I usually wear L, but what if I need an XL (extra large)?

Thankfully, our friends at [Teespring(opens in a new tab)](https://teespring.com/) have got us covered because they've created a sizing chart to make things a lot easier.

### T-Shirt Sizing Chart

| **Size** | **Width** | **Length** | **Sleeve** |
| --- | --- | --- | --- |
| S | 18" | 28" | 8.13" |
| M | 20" | 29" | 8.38" |
| L | 22" | 30" | 8.63" |
| XL | 24" | 31" | 8.88" |
| 2XL | 26" | 33" | 9.63" |
| 3XL | 28" | 34" | 10.13" |

Source: Teespring.com

## Directions:

Use the sizing chart above, create a series of logical expressions that prints the **size** of a t-shirt based on the measurements of shirtWidth, shirtLength, and shirtSleeve. Valid sizes include S, M, L, XL, 2XL, and 3XL.

For example, if...

**const** shirtWidth = 23; *// size L (large)*

**const** shirtLength = 30; *// size L (large)*

**const** shirtSleeve = 8.71; *// size L (large)*

Then print L to the console.

**Hint:** You will need to compare a ***range*** of values when checking for *shirtWidth*, *shirtLength*, and *shirtSleeve*. For example, if the shirt's ***width*** is at least ***20"***, but no more than ***22"***, then the t-shirt should be ***medium (M)*** — as long as the other values for the shirt's ***length*** and ***sleeve*** measurements match up.

If shirtWidth, shirtLength, and shirtSleeve don't fit within the range of acceptable values for a specific **size**, then print NA to the console. For example, if...

**const** shirtWidth = 18; *// size S (small)*

**const** shirtLength = 29; *// size M (medium)*

**const** shirtSleeve = 8.47; *// size M (medium)*

Then print N/A to the console because the measurements don't all match up with one particular size.

### Running Your Code

Enter the following in the terminal:

**node** what-wear.js

### Testing Your Code

How will you know if your code works? Change the values of shirtWidth, shirtLength and shirtSleeve and re-run the code and compare your output with the expected output.

#### Expected Output

| **Width** | **Length** | **Sleeve** | **Size** |
| --- | --- | --- | --- |
| 18 | 28 | 8.13 | S |
| 19.99 | 28.99 | 8.379 | S |
| 20 | 29 | 8.38 | M |
| 22 | 30 | 8.63 | L |
| 24 | 31 | 8.88 | XL |
| 26 | 33 | 9.63 | 2XL |
| 27.99 | 33.99 | 10.129 | 2XL |
| 28 | 34 | 10.13 | 3XL |
| 18 | 29 | 8.47 | NA |

// change the values of ,[object Object],, ,[object Object],, and ,[object Object], to test your code

const shirtWidth = 21;

const shirtLength = 99;

const shirtSleeve = 8.40;

,[object Object],

if ((shirtWidth>=18 && shirtWidth<20) && (shirtLength>=28 && shirtLength<29) && (shirtSleeve>=8.13 && shirtSleeve<8.38) ) {

  size = "S";

  }

  else if ((shirtWidth>=20 && shirtWidth<22) && (shirtLength>=29 && shirtLength<30) && (shirtSleeve>=8.38 && shirtSleeve<8.63) ) {

  size = "M";

  }

  else if ((shirtWidth>=22 && shirtWidth<24) && (shirtLength>=30 && shirtLength<31) && (shirtSleeve>=8.63 && shirtSleeve<8.88) ) {

  size = "L";

  }

  else if ((shirtWidth>=24 && shirtWidth<26) && (shirtLength>=31 && shirtLength<33) && (shirtSleeve>=8.88 && shirtSleeve<9.63) ) {

  size = "XL";

  }

  else if ((shirtWidth>=26 && shirtWidth<28) && (shirtLength>=33 && shirtLength<34) && (shirtSleeve>=9.63 && shirtSleeve<10.13) ) {

  size = "2XL";

  }

  else if ((shirtWidth>=28) && (shirtLength>=34) && (shirtSleeve>=10.13) ) {

  size = "3XL";

  }

  else {

  size = "NA";

  }

  console.log(size);

# Advanced Conditionals

## Beyond The Basics

We've covered the the basics of conditionals and logical operators -- but there is more to learn!

In the rest of this lesson, we'll focus on some more advanced aspects of working with conditional statements:

* Truthy and falsy values
* The ternary operator
* The switch statement

# Truthy and Falsy

## Truthy and Falsy

Every value in JavaScript has an inherent boolean value. When that value is evaluated in the context of a boolean expression, the value will be transformed into that inherent boolean value.

The paragraph above is pretty dense with information. You should probably re-read it again! ☝️

## Falsy values

A value is **falsy** if it converts to false when evaluated in a boolean context. For example, an empty String "" is falsy because, "" evaluates to false. You already know if...else statements, so let's use them to test the truthy-ness of "".

**if** ("") {

**console**.**log**("the value is truthy");

} **else** {

**console**.**log**("the value is falsy");

}

**Returns:** "the value is falsy"

### Here’s the list of all of the falsy values:

1. the Boolean value false

* the null type
* the undefined type
* the number 0
* the empty string ""
* the odd value NaN (stands for "not a number", check out the [NaN MDN article(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN))

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN>

That's right, there are only six falsy values in all of JavaScript!

## Truthy values

A value is **truthy** if it converts to true when evaluated in a boolean context. For example, the number 1 is truthy because, 1 evaluates to true. Let's use an if...else statement again to test this out:

**if** (1) {

**console**.**log**("the value is truthy");

} **else** {

**console**.**log**("the value is falsy");

}

**Returns:** "the value is truthy"

Here are some other examples of truthy values:

true

42

"pizza"

"0"

"null"

"undefined"

{}

[]

**Essentially, if it's not in the list of falsy values, then it's truthy!**

# Ternary Operator

Sometimes, you might find yourself with the following type of conditional.

**const** isGoing = true;

**const** color;

**if** (isGoing) {

color = "green";

} **else** {

color = "red";

}

**console**.**log**(color);

**Prints:** "green"

In this example, the variable color is being assigned to either "green" or "red" based on the value of isGoing. This code works, but it’s a rather lengthy way for assigning a value to a variable. Thankfully, in JavaScript there’s another way.

**TIP:** Using if(isGoing) is the same as using if(isGoing === true). Alternatively, using if(!isGoing) is the same as using if(isGoing === false).

## Ternary operator

The **ternary operator** provides you with a shortcut alternative for writing lengthy if...else statements.

conditional ? (if condition is true) : (if condition is false)

To use the ternary operator, first provide a conditional statement on the left-side of the ?. Then, between the ? and : write the code that would run if the condition is true and on the right-hand side of the : write the code that would run if the condition is false. For example, you can rewrite the example code above as:

**const** isGoing = true;

**const** color = isGoing ? "green" : "red";

**console**.**log**(color);

**Prints:** "green"

This code not only replaces the conditional, but it also handles the variable assignment for color.

If you breakdown the code, the condition isGoing is placed on the left side of the ?. Then, the first expression, after the ?, is what will be run if the condition is true and the second expression after the, :, is what will be run if the condition is false.

### Example

Here's a comparison of using an if...else statement vs. using a ternary operator.

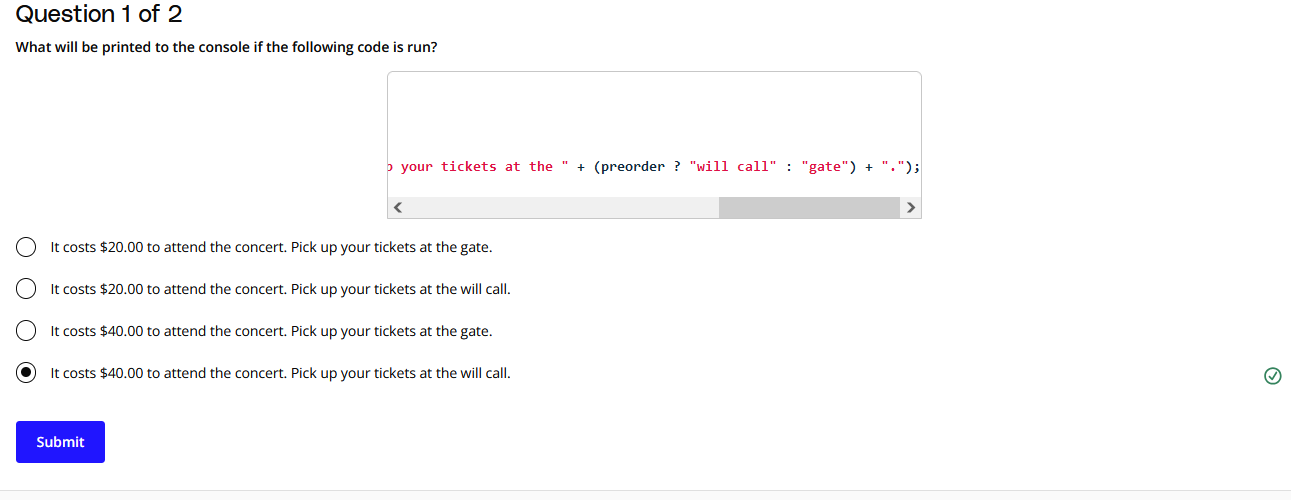
***What is the result of the ternary expression? javascript const x = 4; const y = 3; (x + y) > 6 ? 2* x : 2 *y ;***

const x = 4;

const y = 3;

const result = (x + y) > 6 ? 2 \* x : 2 \* y;

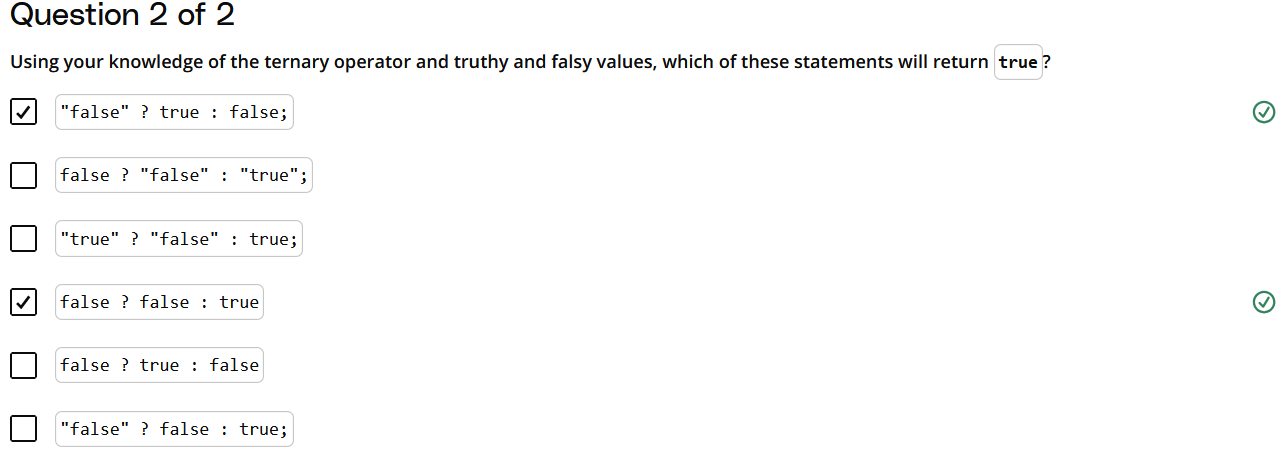
console.log(result);



***const adult = true;***

***const preorder = true;***

***console.log("It costs $" + (adult ? "40.00" : "20.00") + " to attend the concert. Pick up your tickets at the " + (preorder ? "will call" : "gate") + ".");***



### Statements Analysis

1. **false ? "false" : "true"**
   * Since false is falsy, the ternary operator will choose the second option ("true").
   * **Returns:** "true" (which is a truthy value, though it's a string, not a boolean true).
2. **false ? false : true**
   * Since false is falsy, the ternary operator will choose the second option (true).
   * **Returns:** true (this is the boolean true).
3. **"false" ? true : false**
   * "false" is a non-empty string, which is considered truthy in JavaScript.
   * Since "false" is truthy, the ternary operator will choose the first option (true).
   * **Returns:** true (this is the boolean true).
4. **false ? "false" : "true"**
   * This statement is the same as statement #1.
   * **Returns:** "true" (which is a truthy value, though it's a string, not a boolean true).
5. **"false" ? false : true**
   * "false" is a non-empty string, so it is truthy.
   * Since "false" is truthy, the ternary operator will choose the first option (false).
   * **Returns:** false (this is the boolean false).

### Summary of Statements That Return true

The statements that will return true are:

* **false ? false : true** (boolean true)
* **"false" ? true : false** (boolean true)

**Note:** Statements #1 and #4 return "true" (a string), which is truthy, but not a boolean true.

# Quiz: Navigating the Food Chain

From the smallest of creatures to the largest of animals, inevitably every living, breathing thing must ingest other organisms to survive. This means that all animals will fall within one of the three consumer-based categories based on the types of food that they eat.

* Animals that eat only plants are called **herbivores**
* Animals that eat only other animals are called **carnivores**
* Animals that eat both plants and animals are called **omnivores**

## Directions:

Write a series of nested ternary statements that sets the variable category equal to:

* "herbivore" if an animal eats plants
* "carnivore" if an animal eats animals
* "omnivore" if an animal eats plants and animals
* "undefined" if an animal doesn't eat plants or animals

***HINT*** If you get stuck, try drawing a diagram of the decision tree.

#### Running Your Code

Create a file named food-chain.js, implement the solution, and run the command below on your terminal:

**node** food-chain.js

### Testing Your Code

How will you know if your code works? Change the values of eatsPlants and eatsAnimals and re-run the code and compare your output with the expected output.

#### Expected Output

| **Eats Plants** | **Eats Animals** | **Expected Output** |
| --- | --- | --- |
| true | false | "herbivore" |
| true | true | "omnivore" |
| false | true | "carnivore" |
| false | false | undefined |

// Variables to determine what the animal eats

const eatsPlants = true;   // Change this for testing

const eatsAnimals = false; // Change this for testing

// Nested ternary statement to determine the category

const category = eatsPlants

    ? (eatsAnimals ? "omnivore" : "herbivore")

    : (eatsAnimals ? "carnivore" : "undefined");

console.log(category);

[***https://medium.com/javascript-scene/nested-ternaries-are-great-361bddd0f340***](https://medium.com/javascript-scene/nested-ternaries-are-great-361bddd0f340)

# Switch Statement

If you find yourself repeating else if statements in your code, where each condition is based on the same value, like this:

**if** (option === 1) {

**console**.**log**("You selected option 1.");

} **else** **if** (option === 2) {

**console**.**log**("You selected option 2.");

} **else** **if** (option === 3) {

**console**.**log**("You selected option 3.");

} **else** **if** (option === 4) {

**console**.**log**("You selected option 4.");

} **else** **if** (option === 5) {

**console**.**log**("You selected option 5.");

} **else** **if** (option === 6) {

**console**.**log**("You selected option 6.");

}

then it might be time to use a **switch statement**.

## Switch Statement

A **switch statement** is an another way to chain multiple else if statements that are based on the same value **without using conditional statements**. Instead, you just switch which piece of code is executed based on a value.

**switch** (option) {

**case** 1:

**console**.**log**("You selected option 1.");

**case** 2:

**console**.**log**("You selected option 2.");

**case** 3:

**console**.**log**("You selected option 3.");

**case** 4:

**console**.**log**("You selected option 4.");

**case** 5:

**console**.**log**("You selected option 5.");

**case** 6:

**console**.**log**("You selected option 6.");

}

Here, each else if statement (option === [value]) has been replaced with a case clause (case [value]:) and those clauses have been wrapped inside the switch statement.

### Switch Looks for Matching Values

When the switch statement evaluates, it starts at the top and looks for a case clause whose expression evaluates to the same value as the result of the expression passed to the switch statement.

When it finds a match, it transfers control to that case clause to executed the code for that case.

So, if you set option equal to 3...

**const** option = 3;

**switch** (option) {

...

}

**Prints:**  
You selected option 3.  
You selected option 4.  
You selected option 5.  
You selected option 6.

...then the switch statement prints out options 3, 4, 5, and 6.

But that’s not exactly like the original if...else code at the top? So what’s missing?

## Use Break to Avoid Falling Through

After the code in the matching case is run, the switch statement continues to run all of the code below that statement too! This is called **falling through**.

We can prevent the code from falling through by adding a **break statement** at the end of each case.

The break statement will terminate the switch statement and transfer control to the code following the switch statement which prevents the switch statement from falling through and running the code in the other case clauses.

**const** option = 3;

**switch** (option) {

**case** 1:

**console**.**log**("You selected option 1.");

**break**;

**case** 2:

**console**.**log**("You selected option 2.");

**break**;

**case** 3:

**console**.**log**("You selected option 3.");

**break**;

**case** 4:

**console**.**log**("You selected option 4.");

**break**;

**case** 5:

**console**.**log**("You selected option 5.");

**break**;

**case** 6:

**console**.**log**("You selected option 6.");

**break**; *// technically, not needed*

}

**Prints:** You selected option 3.

### Set a Default Case

What happens if none of the cases match? Nothing -- because there is no code to run. That works in some situations, but it most cases you'll want to add a default case to catch situations when none of the case statements match.

**const** option = 23;

**switch** (option) {

**case** 1:

**console**.**log**("You selected option 1.");

**break**;

**case** 2:

**console**.**log**("You selected option 2.");

**break**;

**case** 3:

**console**.**log**("You selected option 3.");

**break**;

**case** 4:

**console**.**log**("You selected option 4.");

**break**;

**case** 5:

**console**.**log**("You selected option 5.");

**break**;

**case** 6:

**console**.**log**("You selected option 6.");

**break**;

**default**:

**console**.**log**("You did not select a valid option.");

}

**Prints:** You did not select a valid option.

It is good practice to ***always*** set a default case.

[***https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch***](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch)

# Leveraging Falling Through

In some situations, you might want to leverage the "falling-through" behavior of switch statements to your advantage.

For example, when your code follows a hierarchical-type structure.

**const** tier = "nsfw deck";

**let** output = "You’ll receive "

**switch** (tier) {

**case** "deck of legends":

output += "a custom card, ";

**case** "collector's deck":

output += "a signed version of the Exploding Kittens deck, ";

**case** "nsfw deck":

output += "one copy of the NSFW (Not Safe for Work) Exploding Kittens card game and ";

**default**:

output += "one copy of the Exploding Kittens card game.";

}

**console**.**log**(output);

**Prints:** You’ll receive one copy of the NSFW (Not Safe for Work) Exploding Kittens card game and one copy of the Exploding Kittens card game.

In this example, based on the [successful Exploding Kittens Kickstarter campaign(opens in a new tab)](https://www.kickstarter.com/projects/elanlee/exploding-kittens/description) (a hilarious card game created by Elan Lee), each successive tier builds on the next by adding more to the output. Without any break statements in the code, after the switch statement jumps to the "nsfw deck", it continues to fall-through until reaching the end of the switch statement.

Also, notice the default case.

**const** tier = "none";

**let** output = "You’ll receive ";

**switch** (tier) {

...

**default**:

output += "one copy of the Exploding Kittens card game.";

}

**console**.**log**(output);

**Prints:** You’ll receive one copy of the Exploding Kittens card game.

By using the falling-through behavior of switch statements, you can represent hierarchical-type scenarios like the Kickstarter backer program. 

# Quiz: Back to School

In 2015, the U.S. Bureau of Labor Statistics [conducted research(opens in a new tab)](http://www.bls.gov/emp/ep_chart_001.htm) to reveal how average salary is directly related to the number of years spent in school. In their findings, they found that people with:

* no high school diploma earned an average of **$25,636/year**,
* a high school diploma earned an average of **$35,256/year**,
* an Associate's degree earned an average of **$41,496/year**,
* a Bachelor's degree earned an average of **$59,124/year**,
* a Master's degree earned an average of **$69,732/year**,
* a Professional degree earned an average of **$89,960/year**,
* and a Doctoral degree earned an average of **$84,396/year**.

**NOTE:** Wondering what the average salary would be for a person with a Nanodegree from Udacity? That's a hard question to answer, but that doesn't mean we haven't tried to quantify the value of our Nanodegrees. [Read more about Nanodegrees from resident Udacity writer, Chris Watkins, here(opens in a new tab)](http://blog.udacity.com/2016/07/nanodegree-101.html)

## Directions:

Write a switch statement to set the average salary of a person based on their type of completed education.

Afterwards, print the following to the console.

In 2015, a person with \_\_\_\_\_\_\_\_\_\_ earned an average of \_\_\_\_\_\_\_\_\_\_/year.

Fill in the blanks with the type of education and the expected average salary. Make sure to use correct grammar in your printed statement, **and watch out for any extra or missing characters** (including spaces and punctuation marks). For help, refer to the findings above.

In 2015, a person with a Bachelor's degree earned an average of $59,124/year.

**TIP:** To print out the average salary with commas (i.e. 59,124), use the toLocaleString() method and pass it the locale "en-US". For example, salary.toLocaleString("en-US").

### Running Your Code

Enter the following in the terminal:

**node** back-to-school.js

### Testing Your Code

How will you know if your code works? Change the value of education and re-run the code.

| **Salary** | **Education** |
| --- | --- |
| $25,636 | no high school diploma |
| $35,256 | high school diploma |
| $41,496 | Associate's degree |
| $59,124 | Bachelor's degree |
| $69,732 | Master's degree |
| $89,960 | Professional degree |
| $84,396 | Doctoral degree |

## Start Workspace

// Switch statement to determine salary based on education

switch (education) {

  case "no high school diploma":

    salary = 25636;

    break;

  case "high school diploma":

    salary = 35256;

    break;

  case "Associate's degree":

    salary = 41496;

    break;

  case "Bachelor's degree":

    salary = 59124;

    break;

  case "Master's degree":

    salary = 69732;

    break;

  case "Professional degree":

    salary = 89960;

    break;

  case "Doctoral degree":

    salary = 84396;

    break;

  default:

    salary = 0; // Default case if education doesn't match any

}

// Print the result to the console

console.log(`In 2015, a person with ${education} earned an average of $${salary.toLocaleString("en-US")}/year.`);

# Lesson Summary

## Key Points to Remember

As you face new challenges and complex problems, remember what you learned in this lesson about using logic to create algorithms:

* Break the problem down into smaller steps
* Use conditional statements and logical operators to tell your code when and how to run

### We Covered A Lot in This Lesson!

You now know how to:

* Write if...else and else...if conditionals
* Use logical operators to handle more complex logic
* Identify truthy and falsy values
* Use ternary operators for more concise conditional logic
* Use a switch statement to chain multiple else if statements

Great work!

# Intro to Loops

## Loops Repeat Blocks of Code

Conditional statements are one way to control the flow of code -- if a certain condition is true, execute this block of code, otherwise, execute that other block of code.

Loops are another way to control the flow of code by allowing us to execute a block of code multiple times.

### What We Will Cover in This Lesson

You will learn how to:

* Use while loops
* Use for loops
* Nest loops for more complex automation
* Use assignment operators to write more concise code

Along the way we'll give you a lot of practice writing loops.

Let's get started!

# while Loops

## Is There a Better Way?

## The Power of Loops

Here is the naive code we wrote to count to 10,000:

**let** x = 1;

**console**.**log**(x + " mississippi!"); *// 1 missippi!*

x = x + 1;

**console**.**log**(x + " mississippi!"); *// 2 missippi!*

x = x + 1;

**console**.**log**(x + " mississippi!"); *// 3 missippi!*

.

.

.

**Prints:**  
1 missippi!  
2 missippi!  
3 missippi!

and so on...

This works, but it is very time consuming. There has to be a better way!

## Loops to the Rescue!

Using a **loop** will let us iterate over values and repeatedly run a block of code.

***Example:*** a while loop

**let** x = 1;

**while** (x <= 10000) {

**console**.**log**(x + " mississippi!");

x = x + 1;

}

### Why does the video use var and the example code use let?

Although var works, it is now considered best practice to use let or const instead. We're showing you both but you should try to use let and const in your own code.

# Parts of a while Loop

## Parts of a while Loop

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times.

Three main pieces of information that any loop should have are:

1. **When to start:** The code that sets up the loop — defining the starting value of a variable for instance.
2. **When to stop:** The logical condition to test whether the loop should continue.
3. **How to get to the next item:** The incrementing or decrementing step — for example, x = x \* 3 or x = x - 1

Here's a basic while loop example that includes all three parts.

**let** start = 0; *// when to start*

**while** (start < 10) { *// when to stop*

**console**.**log**(“start = ”, start);

start = start + 2; *// how to get to the next item*

}

**Prints:**  
start = 0  
start = 2  
start = 4  
start = 6  
start = 8

If a loop is missing any of these three things, then you might find yourself in trouble. For instance, a missing stop condition can result in a loop that never ends!

**⚠️ Don't run this code! ⚠️**

**while** (true) {

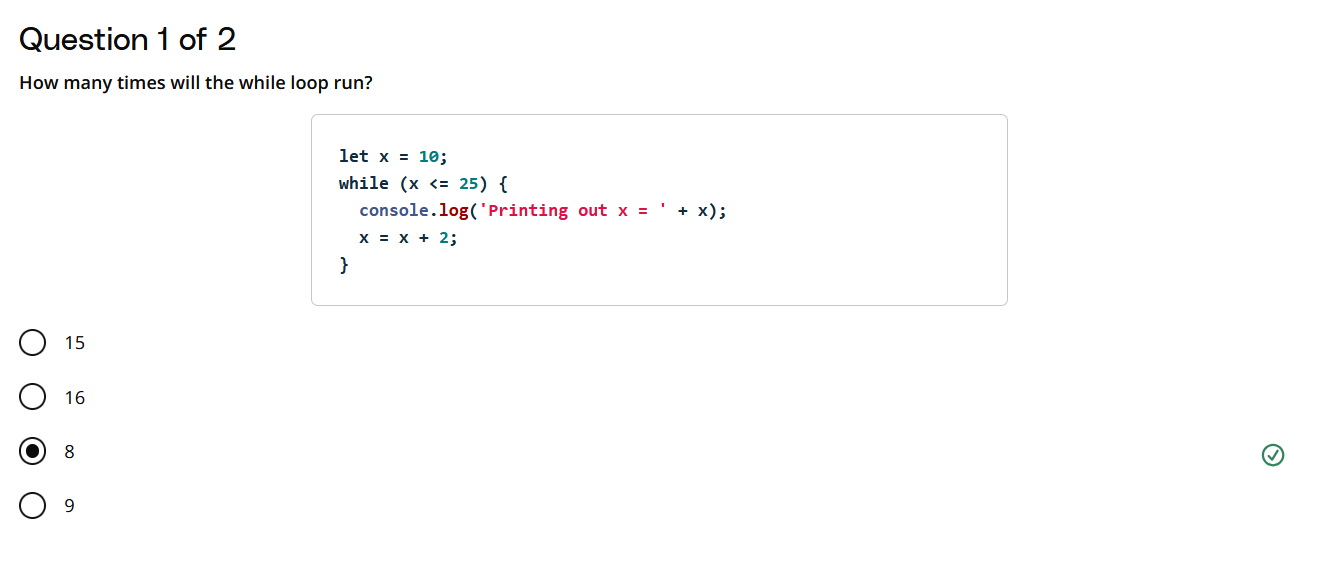
**console**.**log**("true is never false, so I will never stop!");

}

If you did try to run that code in the console, you would probably crash your browser tab.

**WARNING:** You’re probably reading this because you didn't heed our warnings about running that infinite loop in the console. If your browser tab has crashed or has become frozen/unresponsive, there are a couple ways to fix this. If you are using Firefox, the browser will popup a notification about your script being unresponsive, and will give you the option to kill the script (do that). If you're using Chrome, go to the taskbar and select Window > Task Manager. You can end the process for the particular tab you ran the script in through the task manager. If you’re not using Firefox or Chrome, download Firefox or Chrome ;).

An infinite loop will run forever... until your browser crashes or you stop this video 😉



// Print the result to the console

console.log(`In 2015, a person with ${education} earned an average of $${salary.toLocaleString("en-US")}/year.`);

let x = 10;

while (x <= 25) {

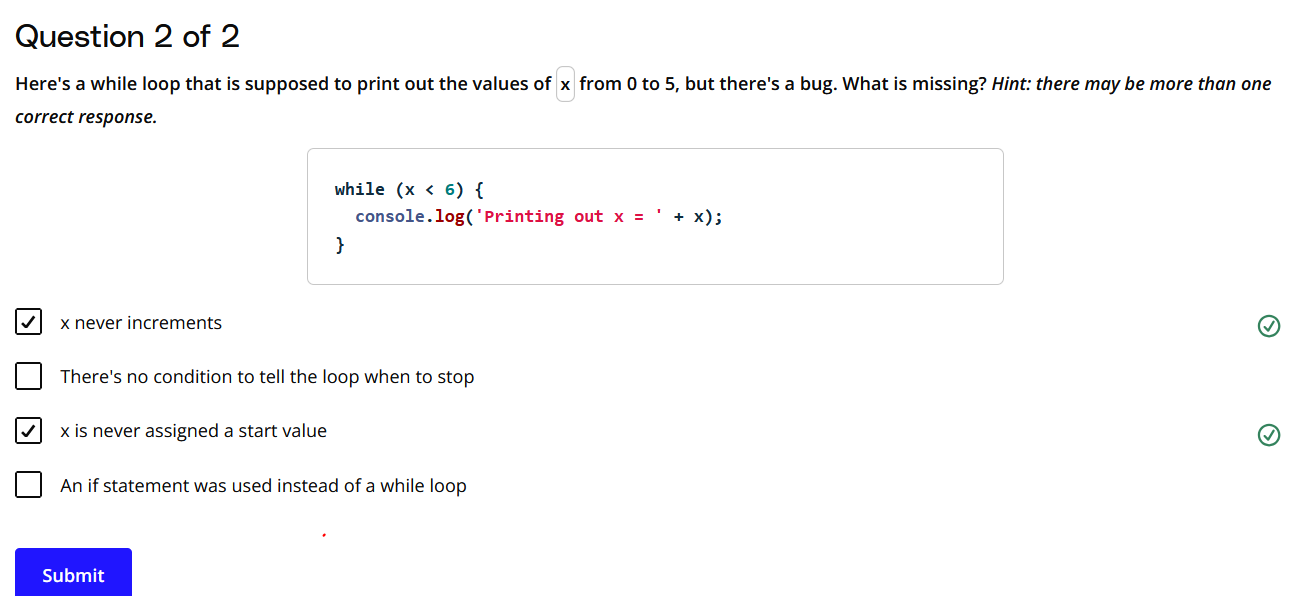
  console.log('Printing out x = ' + x);

  x = x + 2;

}

### Loop Execution Steps

1. **Initialization**: The variable x starts at 10.
2. **First Iteration**:
   * Condition: 10 <= 25 (true)
   * Print: Printing out x = 10
   * Update: x = 10 + 2 → x = 12
3. **Second Iteration**:
   * Condition: 12 <= 25 (true)
   * Print: Printing out x = 12
   * Update: x = 12 + 2 → x = 14
4. **Third Iteration**:
   * Condition: 14 <= 25 (true)
   * Print: Printing out x = 14
   * Update: x = 14 + 2 → x = 16
5. **Fourth Iteration**:
   * Condition: 16 <= 25 (true)
   * Print: Printing out x = 16
   * Update: x = 16 + 2 → x = 18
6. **Fifth Iteration**:
   * Condition: 18 <= 25 (true)
   * Print: Printing out x = 18
   * Update: x = 18 + 2 → x = 20
7. **Sixth Iteration**:
   * Condition: 20 <= 25 (true)
   * Print: Printing out x = 20
   * Update: x = 20 + 2 → x = 22
8. **Seventh Iteration**:
   * Condition: 22 <= 25 (true)
   * Print: Printing out x = 22
   * Update: x = 22 + 2 → x = 24
9. **Eighth Iteration**:
   * Condition: 24 <= 25 (true)
   * Print: Printing out x = 24
   * Update: x = 24 + 2 → x = 26
10. **Ninth Iteration**:
    * Condition: 26 <= 25 (false)



[***https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while***](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while)

# Quiz: JuliaJames

"Fizzbuzz" is a famous interview question used in programming interviews. It goes something like this:

* Loop through the numbers 1 to 100
* If the number is divisible by 3, print "Fizz"
* If the number is divisible by 5, print "Buzz"
* If the number is divisible by both 3 and 5, print "FizzBuzz"
* If the number is **not** divisible by 3 or 5, print the number

**TIP:** A number x is divisible by a number y if the answer to x / y has a remainder of 0. For example, 10 is divisible by 2 because 10 / 2 = 5 with no remainder. You can check if a number is divisible by another number by checking if x % y === 0.

We're going to have you program your own version of FizzBuzz called "JuliaJames" (yes, imaginative, right?) Keep in mind that in an interview, you would want to write efficient code with very little duplication. We don't want you to worry about that for this question. Just focus on practicing using loops.

## Directions:

Write a while loop that:

* Loop through the numbers 1 to 20
* If the number is divisible by 3, print "Julia"
* If the number is divisible by 5, print "James"
* If the number is divisible by 3 and 5, print "JuliaJames"
* If the number is **not** divisible by 3 or 5, print the number

### Running Your Code

Enter the following in the terminal:

**node** julia-james.js

[Node.js(opens in a new tab)](https://nodejs.org/en/about/) is an open-source backend JavaScript runtime environment. That basically means that it will run your JavaScript code in the terminal and return any output that is generated. We'll be using Node to run the code in our workspaces for many of the quizzes in this course.

### Testing Your Code

Your code should log out the following in the terminal:

1

2

Julia

4

James

Julia

7

8

Julia

James

11

Julia

13

14

JuliaJames

16

17

Julia

19

James

let number = 1; // Initialize the starting number

while (number <= 20) { // Loop until number is 20

  if (number % 3 === 0 && number % 5 === 0) {

    console.log("JuliaJames"); // Divisible by both 3 and 5

  } else if (number % 3 === 0) {

    console.log("Julia"); // Divisible by 3

  } else if (number % 5 === 0) {

    console.log("James"); // Divisible by 5

  } else {

    console.log(number); // Not divisible by 3 or 5

  }

  number++; // Increment number to move to the next iteration

}

# Quiz: 99 Bottles of Juice

## Directions:

Write a loop that prints out the following song. Starting at 99, and ending at 1 bottle.

99 bottles of juice on the wall! 99 bottles of juice! Take one down, pass it around... 98 bottles of juice on the wall!

98 bottles of juice on the wall! 98 bottles of juice! Take one down, pass it around... 97 bottles of juice on the wall!

...

2 bottles of juice on the wall! 2 bottles of juice! Take one down, pass it around... 1 bottle of juice on the wall!

1 bottle of juice on the wall! 1 bottle of juice! Take one down, pass it around... 0 bottles of juice on the wall!

### Attention to Detail is Important in Coding!

Pay attention to the pluralization of the word "bottle" when you go from 2 bottles to 1 bottle to 0 bottles.

It may not seem critical in this code, but what if you were coding a message to be sent out to important customers? Using incorrect pluralization might cause them to question your organization's competence and ability to meet their needs.

let num = 99;

while (num >=1) {

    // Last iteration. Note occurances of bottle, bottle, bottleS

    if (num === 1) {

        console.log(num + " bottle of juice on the wall! "

                    + num + " bottle of juice! Take one down, pass it around... "

                    + (num-1) + " bottles of juice on the wall!");

    }

    // Second-last iteration. Note occurances of bottleS, bottleS, bottle

    else if (num === 2){

        console.log(num + " bottles of juice on the wall! "

                    + num + " bottles of juice! Take one down, pass it around... "

                    + (num-1) + " bottle of juice on the wall!");

    }

    // All other iterations. Note occurances of bottleS, bottleS, bottleS

    else {

        console.log(num + " bottles of juice on the wall! "

                    + num + " bottles of juice! Take one down, pass it around... "

                    + (num-1) + " bottles of juice on the wall!");

    }

    num = num - 1;

}

# Quiz: Countdown, Liftoff!

NASA's countdown to launch [includes checkpoints(opens in a new tab)](http://www.nasa.gov/mission_pages/shuttle/launch/countdown101.html) where NASA engineers complete certain technical tasks. During the final minute, NASA has 6 tasks to complete:

* Orbiter transfers from ground to internal power (T-50 seconds)
* Ground launch sequencer is go for auto sequence start (T-31 seconds)
* Activate launch pad sound suppression system (T-16 seconds)
* Activate main engine hydrogen burnoff system (T-10 seconds)
* Main engine start (T-6 seconds)
* Solid rocket booster ignition and liftoff! (T-0 seconds)

**NOTE:** "T-50 seconds" read as "T-minus 50 seconds".

**Directions:**

Write a while loop that counts down from 60 seconds and:

* If there's a task being completed, it prints out the task
* If there is no task being completed, it prints out the time as T-x seconds

Use the task and time descriptions described above.

// Let us take an iteration variable, that represent the remaining Time in seconds

let n = 60

// While loop with a stop condition

while (n >= 0) {

    if (n === 50) {

        console.log("Orbiter transfers from ground to internal power");

    }

    else if (n === 31) {

        console.log("Ground launch sequencer is go for auto sequence start");

    }

    else if (n === 16) {

        console.log("Activate launch pad sound suppression system");

    }

    else if (n === 10) {

        console.log("Activate main engine hydrogen burnoff system");

    }

    else if (n === 6) {

        console.log("Main engine start");

    }

    else if (n === 0) {

        console.log("Solid rocket booster ignition and liftoff!");

    }

    else{

        console.log("T-"+n+" seconds");

    }

    //Never forget to decrement/increment the iteration variable in a while loop

    // Otherwise, you loop will run infinite iterations

    n = n-1;

}

# for Loops

## Loop Basics

A loop should always include:

* Where to start
* When to stop
* How to get to the next item

It's easy to forget some of these pieces in a while loop and end up with an infinte loop that crashes your browser!

for loops give you more control over the looping process.

# Parts of a for Loop

## for Loops Require a Start, Stop and Step

The for loop explicitly forces you to define the start point, stop point, and each step of the loop. In fact, you'll get an Uncaught SyntaxError: Unexpected token ) if you leave out any of the three required pieces.

**for** ( start; stop; step ) {

*// do this thing*

}

Here's an example of a for loop that prints out the values from 0 to 5. Notice the semicolons separating the different statements of the for loop: var i = 0; i < 6; i = i + 1

**for** (**let** i = 0; i < 6; i = i + 1) {

**console**.**log**("Printing out i = " + i);

}

**Prints**:  
Printing out i = 0  
Printing out i = 1  
Printing out i = 2  
Printing out i = 3  
Printing out i = 4  
Printing out i = 5

### How the Loop is Interpreted

| **i** | **i < 6** | **console.log output** |
| --- | --- | --- |
| 0 | true | Printing out i = 0 |
| 1 | true | Printing out i = 1 |
| 2 | true | Printing out i = 2 |
| 3 | true | Printing out i = 3 |
| 4 | true | Printing out i = 4 |
| 5 | true | Printing out i = 4 |
| 6 | false |  |

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for>

# Nested Loops

## Nested Loops Add Complexity

**Nested loops** are just loops inside of other loops. Take a look at our demo code:

**for** (**let** x = 0; x < 3; x = x + 1) {

**for** (**let** y = 0; y < 2; y = y + 1) {

**console**.**log**(x + ", " + y);

}

}

This is how the loop is interpreted in the browser:

| **x** | **x < 3** | **y** | **y < 2** | **console.log output** |
| --- | --- | --- | --- | --- |
| 0 | true | 0 | true | 0, 0 |
| 0 | true | 1 | true | 0, 1 |
| 0 | true | 2 | false |  |
| 1 | true | 0 | true | 1, 0 |
| 1 | true | 1 | true | 1, 1 |
| 1 | true | 2 | false |  |
| 2 | true | 0 | true | 2, 0 |
| 2 | true | 1 | true | 2, 1 |
| 2 | true | 2 | false |  |
| 3 | false |  |  |  |

### Try It Yourself!

Paste this nested loop in your browser and take a look at what it prints out:

**for** (**let** x = 0; x < 5; x = x + 1) {

**for** (**let** y = 0; y < 3; y = y + 1) {

**console**.**log**(x + "," + y);

}

}

**Prints**:  
0, 0  
0, 1  
0, 2  
1, 0  
1, 1  
1, 2  
2, 0  
2, 1  
2, 2  
3, 0  
3, 1  
3, 2  
4, 0  
4, 1  
4, 2

Notice the order that the output is being displayed.

For each value of x in the outer loop, the inner for loop executes completely. The outer loop starts with x = 0, and then the inner loop completes its cycle with all values of y:

x = 0 and y = 0, 1, 2 // corresponds to (0, 0), (0, 1), and (0, 2)

Once the inner loop is done iterating over y, then the outer loop continues to the next value, x = 1, and the whole process begins again.

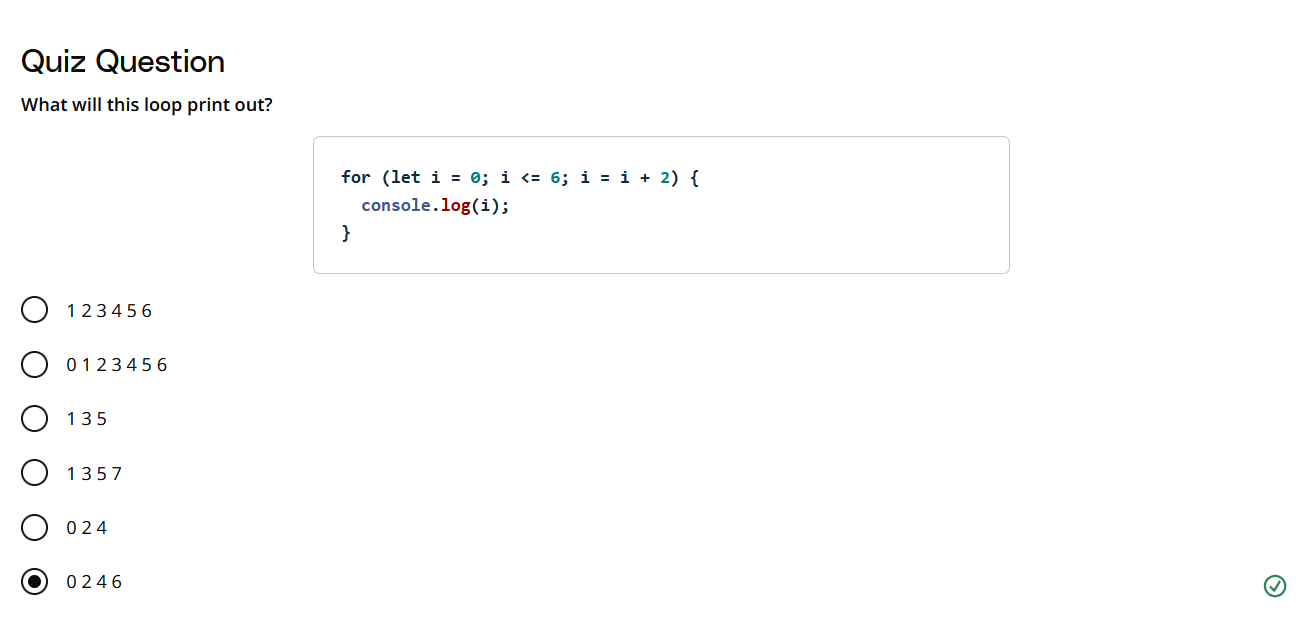
x = 0 and y = 0, 1, 2 // (0, 0) (0, 1) and (0, 2)

x = 1 and y = 0, 1, 2 // (1, 0) (1, 1) and (1, 2)

x = 2 and y = 0, 1, 2 // (2, 0) (2, 1) and (2, 2)

etc.

**NOTE:** Nested loops can be tricky at first. We will revisit them again when we talk about **arrays**, which are lists of data. Loops are very helpful when working with arrays.



# Operators

## Increment, Decrement and Assignment Operators

## Increment and Decrement Operators

Increment and decrement operators are shortcuts that are often used in the step part of a for loop.

### Increment Operator

The **increment operator** ++ adds one a to variable, returns a value and assigns the incremented value to the variable.

x++ is the **postfix** operator, which means that it returns the value before incrementing it:

**let** x = 2;

x++ *//returns 2 then assigns 3 as the value of x*

**console**.**log**(x); *// logs out 3*

++x is the **prefix** operator, which means that it returns the value after incrementing it:

**let** x = 2;

++x *// assigns 31 as the value of x then returns 3*

**console**.**log**(x); *// logs out 3*

Try it in the the console!

### Decrement Operator

a The **decrement operator** -- subtracts one from a variable, returns a value and assigns the decremented value to the variable.

Similiar to the increment operator, x-- is the **postfix** operator, which means that it returns the value before incrementing it:

**let** x = 2;

x-- *//returns 2 then assigns 1 as the value of x*

**console**.**log**(x); *// logs out 1*

--x is the **prefix** operator, which means that it returns the value after incrementing it:

**let** x = 2;

--x *// assigns 1 as the value of x then returns 1*

**console**.**log**(x); *// logs out 1*

Try it in the the console!

## Assignment Operators

An **assignment operator** is a shorthand way to peform a mathematical operation on a variable and assigns that value to the variable.

You can use assignment operators for addition, subtraction, multiplication, and division.

*// Add y to x*

x += y *// x = x + y*

*// Subtract y from x*

x -= y *// x = x - y*

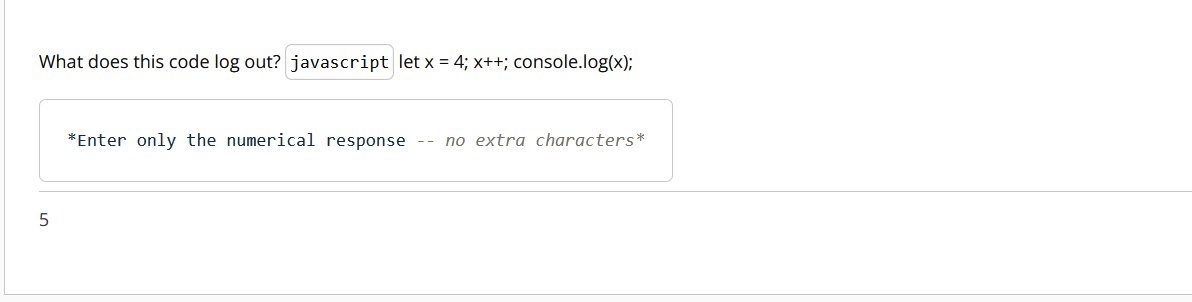
*// Multiply x by x*

x \*= y *// x = x\* y*

*// Divide x by y*

x /= y *// x = x / y*

These assignment operators will come in handy as you create more loops!



let x = 4;  // Step 1: Declare a variable 'x' and assign it the value of 4.

x++;        // Step 2: Use the increment operator to increase the value of 'x' by 1.

console.log(x); // Step 3: Log the value of 'x' to the console.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#increment_and_decrement>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#assignment_operators>

# Quiz: Changing the Loop

## Directions:

Rewrite the following while loop as a for loop:

**let** x = 9;

**while** (x >= 1) {

**console**.**log**("hello " + x);

x = x - 1;

}

### Running Your Code

Enter the following in the terminal:

**node** change.js

### Testing Your Code

Make sure you are using a for loop and that you get this output:

hello 9

hello 8

hello 7

hello 6

hello 5

hello 4

hello 3

hello 2

hello 1

for (let x = 9; x>=1; x--){

  console.log("hello " + x);

}

# Quiz: Fix the Error 1

## Directions:

Here is a for loop that's supposed to print the numbers 5 through 9. Fix the errors!

**for** (x < 10; x++) {

**console**.**log**(x);

}

for (let x = 5; x < 10; x++) {

  console.log(x);

}

# Quiz: Fix the Error 2 (4-6)

## Directions:

The for loop below has an error. Fix it!

**for** (**let** k = 12 k < 21 k++) {

**console**.**log**(k);

}

// The semicolon was missing at two places

for (let k = 12; k < 21; k++) {

  console.log(k);

}

# Quiz: Factorials! (4-7)

## irections:

Write a for (note: not a function) loop that prints out the factorial of the number 12:

A **factorial** is calculated by multiplying a number by all the numbers below it. For instance, 3! or "3 factorial" is 3 \ \*2 \* 1 = 6

3!=3∗2∗1=63!=3∗2∗1=6  
4!=4∗3∗2∗1=244!=4∗3∗2∗1=24  
5!=5∗4∗3∗2∗1=1205!=5∗4∗3∗2∗1=120

Save your final answer in a variable called solution and print it to the console.

let solution = 1;

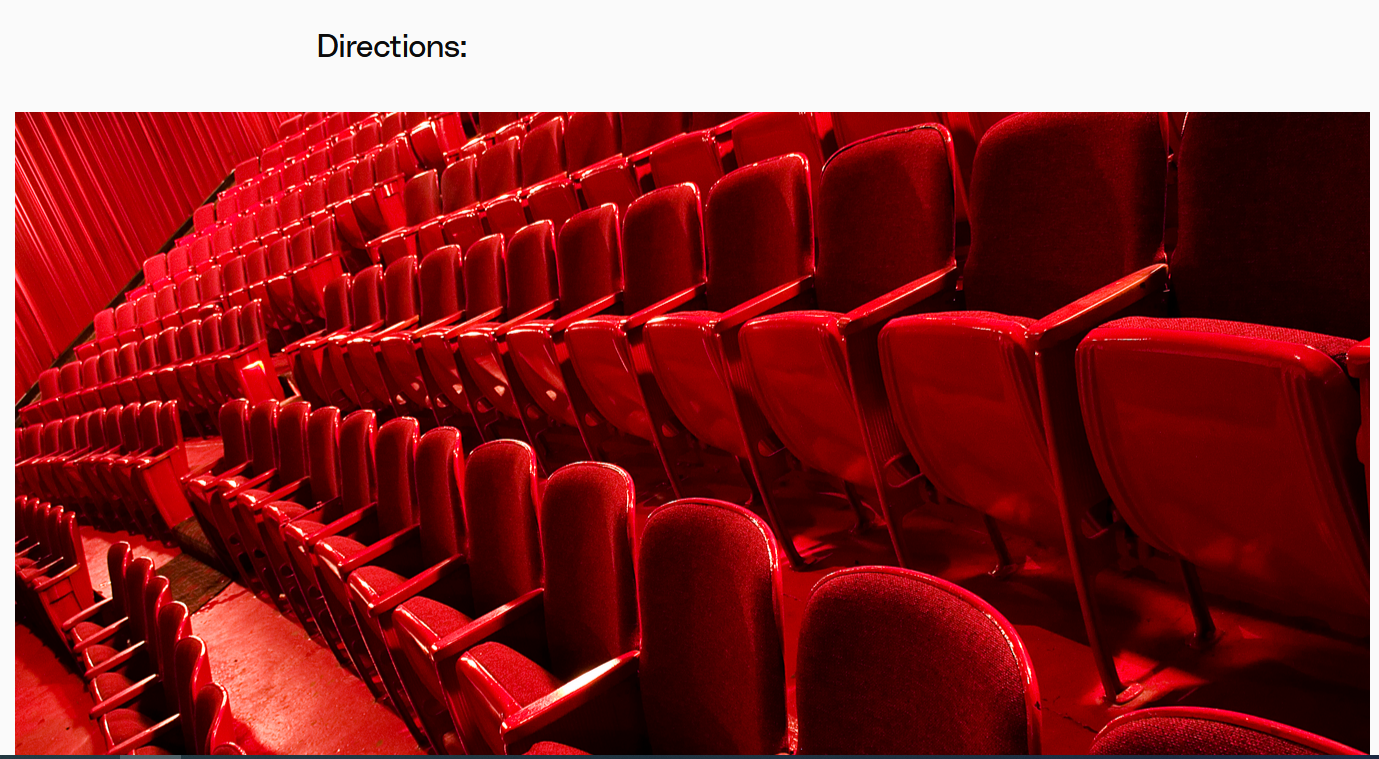
for (let i = 1; i <= 12; i++) {

  solution \*= i;

}

console.log(solution);

# Quiz: Find my Seat (4-8)



Theater seats often display a row and seat number to help theatergoers find their seats. If there are 26 rows (0 to 25) and 100 seats (0 to 99) in each row, write a nested for loop to print out all of the different seat combinations in the theater.

**Example output for row-seat information:** output each row and seat number on a separate line

0-0

0-1

0-2

...

25-97

25-98

25-99

let row = 0;  // initial value of the row

let seat = 0; // initial value of the seat within a row

// One loop inside another is called Nested loop.

// Outer `for` loop, to iterate over the rows

for (row = 0; row <= 25; row++){

    // Inner `for` loop, to iterate over the seats within a row

    // In this loop, the value of `row` variable would change only after 100 iterations

    for(seat = 0; seat <= 99; seat++){

        console.log(row+"-"+seat);

    }

}

# Lesson Summary

## Loops Are a Fundamental Programming Tool

You will be using loops all the time as you progress as a coder. And you should now be really good at them!

### What We Learned in This Lesson

You now know how to:

* Use while loops
* Use for loops
* Nest loops for more complex automation
* Use assignment operators to write more concise code

Next up... functions!

# Intro to Functions

## What Are Functions?

**Functions** are reusable chunks of code.

Functions are very helpful as the problems we need to solve with our code get more complex. We'll often need to repeat steps on different inputs -- and functions let us do that!

\*\*\*Analogy: Microwaving pizza\*\*\*

The microwave ***Pizza Reheat*** button uses a function to calculate the optimal power level and time to reheat pizza so James doesn't have to calculate the settings from scratch every time he is hungry.

* The ***input*** is the number of slices that need to be reheated
* The ***output*** is reheated pizza

### What We Will Cover in This Lesson

You will learn how to:

* Declare and run functions
* Use return statements to return items when a function runs
* Explain the difference between parameters and arguments
* Explain the difference between returning and logging
* Understand how scope works in JavaScript
* Use let and const in block scope
* Understand how hosting works
* Store functions in variables using function expressions
* Use inline function expressions

Let's get started!

# Function Example

## Functions Are Awesome!

The ability to generalize code for a variety of possible inputs is a powerful tool when creating easy to understand, non-repetitive code.

**function** **reverseString**(reverseMe) {

**let** reversed = "";

**for** (**let** i = reverseMe.length - 1; i >= 0; i--) {

reversed += reverseMe[i];

}

**return** reversed;

}

**console**.**log**(**reverseString**("Julia"));

**Prints** "ailuJ"

Let's break it down:

* The function has one ***parameter*** -- a variable named reverseMe.
* reverseMewill store the ***argument*** -- the value of the string that we want the function to operate on.
* The variable reversed is intialized as an empty string. It will be used to store the reversed string as as it is being constructed.
* The function loops through each character the reverseMe string using string indexes, from the end to the beginning and adds each character to reversed.
* When the loop is complete, reversed is returned.

### Annotated Function

*// Set one parameter to hold the value of the input string*

**function** **reverseString**(reverseMe) {

*// Declare a variable with an empty string to store the reversed string*

**let** reversed = "";

*// Loop through the `reverseMe` string from back to front*

**for** (**let** i = reverseMe.length - 1; i >= 0; i--) {

*// Add each character to the end of `reversed`*

reversed += reverseMe[i];

}

**return** reversed;

}

*// Return the completed string when the loop is complete return reversed; }*

**console**.**log**(**reverseString**("Julia"));

Using a function simplifies the process and allows you to reuse the function by calling it by its name and passing it in a string. If these steps were not wrapped inside this reverseString function you would to write all of this code each time you needed to reverse a string.

# Declaring Functions

## How to Declare a Function

**Functions** allow you to package up lines of code that you can use (and often reuse) in your programs.

Sometimes they take **parameters** like the pizza button from the beginning of this lesson. reheatPizza() had one parameter: the number of slices.

**function** **reheatPizza**(numSlices) {

*// code that figures out reheat settings!*

}

The reverseString() function that you saw also had one parameter: the string to be reversed.

**function** **reverseString**(reverseMe) {

*// code to reverse a string!*

}

In both cases, the parameter is listed as a variable after the function name, inside the parentheses. And, if there were multiple parameters, you would just separate them with commas.

**function** **doubleGreeting**(name, otherName) {

*// code to greet two people!*

}

But, you can also have functions that don't have any parameters. Instead, they just package up some code and perform some task. In this case, you would just leave the parentheses empty. Take this one for example. Here's a simple function that just prints out "Hello!".

*// accepts no parameters! parentheses are empty*

**function** **sayHello**() {

**const** message = "Hello!"

**console**.**log**(message);

}

If you tried pasting any of the functions above into the JavaScript console, you probably didn't notice much happen. In fact, you probably saw undefined returned back to you. undefined is the default return value on the console when nothing is explicitly returned using the special return keyword.

### Return Statements

In the sayHello() function above, a value is **printed** to the console with console.log, but not explicitly returned with a **return statement**. You can write a return statement by using the return keyword followed by the expression or value that you want to return.

*// declares the sayHello function*

**function** **sayHello**() {

**const** message = "Hello!"

**return** message; *// returns value instead of printing it*

}

## How to Run a function

Now, to get your function to do something, you have to **invoke** or **call** the function using the function name, followed by parentheses with any **arguments** that are passed into it. Functions are like machines. You can build the machine, but it won't do anything unless you also turn it on. Here's how you would call the sayHello() function from before, and then use the return value to print to the console:

*// declares the sayHello function*

**function** **sayHello**() {

**const** message = "Hello!"

**return** message; *// returns value instead of printing it*

}

*// function returns "Hello!" and console.log prints the return value*

**console**.**log**(**sayHello**());

**Prints:** "Hello!"

## Parameters vs. Arguments

At first, it can be a bit tricky to know when something is either a parameter or an argument. The key difference is in where they show up in the code. A **parameter** is always going to be a variable name and appears in the function declaration. On the other hand, an **argument** is always going to be a value (i.e. any of the JavaScript data types - a number, a string, a boolean, etc.) and will always appear in the code when the function is called or invoked.

Try declaring and calling some functions on your own:

# Function Recap

## What You’ve Learned So Far:

* **Functions** package up code so you can easily use (and reuse) a block of code.
* **Parameters** are variables that are used to store the data that's passed into a function for the function to use.
* **Arguments** are the actual data that's passed into a function when it is invoked:

*// x and y are parameters in this function declaration*

**function** **add**(x, y) {

*// function body*

*// Here, `sum` variable has a scope within the function.*

*// Such variables defined within a function are called Local variables*

*// You can try giving it another name*

**const** sum = x + y;

**return** sum; *// return statement*

}

*// 1 and 2 are passed into the function as arguments,*

*// and the result returned by the function is stored in a new variable `sum`*

*// Here, `sum` is another variable, different from the one used inside the function*

**const** sum = **add**(1, 2);

The **function body** is enclosed inside curly brackets:

**function** **add**(x, y) {

*// function body!*

}

**Return statements** explicitly make your function return a value:

**return** sum;

You **invoke** or **call** a function to have it do something:

**add**(1, 2);

**Returns:** 3

# Quiz: Laugh it Off 1

## Directions:

1. Declare a function called laugh() that returns "hahahahahahahahahaha!".
2. Print the value returned from the laugh() function to the console. It should print:

hahahahahahahahahaha!

// your code goes here

function laugh() {

  // create a variable with any name of your choice

  const sound = "hahahahahahahahahaha!"

  return sound;

}

// test your code by logging out the returned value

// Don't forget to add the () after function name to call it!

console.log(laugh());

# Quiz: Laugh it Off 2

## Directions:

1. Write a function called laugh() that takes one parameter, num.
2. The function should return a string with num number of "ha"s.
3. The string should end with an exclamation point "!".

**TIP:** You might need a loop to solve this!

Here's an example of the output and how to call the function that you will write:

**console**.**log**(**laugh**(3));

**Prints:** "hahaha!"

function laugh(num) {

  let laughString = ""; // Initialize an empty string to build the "ha"s

  for (let i = 0; i < num; i++) {

      laughString += "ha"; // Add "ha" to the string for each loop iteration

  }

  return laughString + "!"; // Append an exclamation point at the end

}

// Example usage

console.log(laugh(3)); // Output: "hahaha!"

**Return Values**

## Test Your Intuition

Before watching the video below, look at the code in the quizzes below. What do you think will be printed when the code is run?

Don't worry if your guess is wrong! We'll explore what is happening with logging and return values in the video.

## Output from a Function

There are two ways to get output from a function:

1. console.log is used to ***print*** a value to the JavaScript console.
2. The return keyword is used to stop execution of a function and \***return** the value back to the caller.

### Points to Remember About Returning and Printing

* **Returning is different from printing**

Printing a value to the JavaScript console only displays the value but the value can't be used anywhere else.

* **Printing is great for debugging code**

Using console.log to test your code in the JavaScript console or to print out values as your code runs can be extremely helpful in pinpointing where something has gone wrong in your code.

* **All function calls return something**

If a return value is not specified, the function will return undefined.

* **The** **return** **keyword will stop the execution of a function**

Any code after a return statement will be ignored.

***Demo Code:*** Function to determine if a number is a prime number

**Math Review**:  
A **prime** number is an integer that is not a product of two smaller integers.   
The [modulo operator(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Remainder) % returns the remainder left over when one operand is divided by a second operand.  
You can check for prime numbers by dividing them by smaller integers. If the number can be divided without remainder by any integer greater than 1 it is not a prime number.

**function** **isPrime**(integer) {

**for** (**let** x = 2; x < integer; x++ ) {

**if**(integer % x === 0) {

**console**.**log**(integer + " is divisible by " + x);

**return** false

}

}

**return** true

}

## Exploring console.log and return Statements

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Remainder>

Paste the following function declaration and function invocation into the JavaScript console to see the difference between logging (printing) and returning:

**function** **isThisWorking**(input) {

**console**.**log**("Printing: isThisWorking was called and " + input + " was passed in as an argument.");

**return** "Returning: I am returning this string!";

}

**isThisWorking**(3);

**Prints:** "Printing: isThisWorking was called and 3 was passed in as an argument"  
**Returns:** "Returning: I am returning this string!"

If you don't explicitly define a return value, the function will return undefined by default.

**function** **isThisWorking**(input) {

**console**.**log**("Printing: isThisWorking was called and " + input + " was passed in as an argument.");

}

**isThisWorking**(3);

**Prints:** "Printing: isThisWorking was called and 3 was passed in as an argument"  
**Returns:** undefined

# Using Return Values

## Using Return Values

Returning a value from a function is great, but what's the use of a return value if you're not going to use the value to do something?

A function's **return value** can be stored in a variable or reused throughout your program as a function argument.

### Example

Here, we have a function that adds two numbers together, and another function that divides a number by 2. We can find the average of 5 and 7 by using the add() function to add a pair of numbers together, and then by passing the sum of the two numbers add(5, 7) into the function divideByTwo() as an argument.

And finally, we can even store the final answer in a variable called average and use the variable to perform even more calculations in more places!

*// returns the sum of two numbers*

**function** **add**(x, y) {

**return** x + y;

}

*// returns the value of a number divided by 2*

**function** **divideByTwo**(num) {

**return** num / 2;

}

**const** sum = **add**(5, 7); *// call the "add" function and store the returned value in the "sum" variable*

**const** average = **divideByTwo**(sum); *// call the "divideByTwo" function*

# Scope

## Understanding Scope

**Scope** is the part of the program where a particular identifier, such as a variable or a function name, is visible or accessible.

As a programmer, you'll run into a lot of situations where understanding scope will be critical to writing effective and error-free code.

\*\*\*Analogy: Finding a Book\*\*\*

When James is inside the library he is **in scope** to find out where the book Great Expectations is shelved. The librarian has that information is able to share it with James.

When James is outside of the library, the person he asked about Great Expectations didn't have any information about library or the books inside of it. In this case, James's question is **out of scope**.

# Global and Function Scope

## he Three Types of Scope in JavaScript

JavaScript has three types of scope.

* global scope
* function scope
* block scope

The first two are pretty simple:

* Variables declared in the **global scope** are accessible to any other part of the program.
* Variables declared inside a function are in the **function scope** which means they are only accessible inside that function.

Block scope is more complicated. We'll learn more about that later in this lesson.

### Translating the Library Example to Code

Let's translate James's requests for a book into JavaScript code so we can take a closer look at the scope:

**const** bookSeeker = "James";

**const** book = "Great Expectations";

**function** **library**() {

**const** librarian = "Julia";

**console**.**log**(bookSeeker + " asked " + librarian + " for " + book);

**function** **classicLiterature**() {

**const** shelf = "Dickens";

**console**.**log**( bookSeeker + " found " + book + " on the " + shelf + " shelf!");

}

**classicLiterature**();

}

### Global Scope

bookSeeker and book are in the global scope so as expected, we can use access them anywhere in the code.

For example, this code runs without an error:

**console**.**log**(bookSeeker + " is looking for " + book);

**Prints** James is looking for Great Expectations

We can also run the library function without an error:

**library**();

**Prints**  
James asked Julia for Great Expectations  
James found Great Expectations on the Dickens shelf!

### Function Scope

What happens when we try to access the librarian variable declared inside the library function from the console? We get an error, because librarian can only be accessed inside the scope of the library function.

**console**.**log**(librarian);

**Returns** Uncaught ReferenceError: librarian is not defined

We get a similar error when we try to access shelf in the console because the shelf variable is declared inside the classicLiterature function.

**console**.**log**(shelf);

**Returns** Uncaught ReferenceError: shelf is not defined

# The Scope Chain

## ow Does Javascript Find a Variable? It Uses the Scope Chain

When the JavaScript engine is looking for a variable, it starts from the current scope and moves outward:

1. The JavaScript engine will start looking in the scope where the variable is requested.
2. If it can't find it in the current scope, it moves out one level and checks again.
3. It keeps moving to the outer scope until it has reached the global scope.
4. If the JavaScript engine checks all of the outer functions and global scope, and it still can't find the identifier then it will return a Reference error.

*// <-- 4. JavaScript engine looks here last*

**const** globalVar = "I am in the global scope";

**function** **outerOuterFunction**() {

*// <-- 3. JavaScript engine looks here third*

**const** outerOuterVar = 'I am in the outerOuterFunction scope';

**function** **outerFunction**() {

*// <-- 2. JavaScript engine looks here second*

**const** outerVar = 'I am in the outerFunction scope';

**function** **innerFunction**() {

*// <-- 1. JavaScript engine looks here first*

**const** innerVar = 'I am in the innerFunction scope';

**console**.**log**(globarVar);

}

}

}

# Block Scope

## A New Type of Scope

With the ES6 version of JavaScript, a new type of scope was created: **block scope** which limits the scope of a variable to the block of code where it is declared.

### What is a Block?

A **block** is a group of statements in between curly braces. You've seen blocks in conditional statements:

**const** x = 5;

**if** (x < 6) {

**const** double = x \* 2;

**console**.**log**(double);

} **else** {

**const** half = x / 2;

**console**.**log**(half);

}

and in loops:

**for** (**let** i = 0; i < 5; i++) {

**let** triple = x \* 3;

**console**.**log**(triple);

}

In the examples above, the variables double, half and triple are only available inside the block where they are declared.

### Block Scope Only Works with let And const

Unlike with function scope, if you declare a variable inside a block using var the variable will be accessible both inside the block and in the block's outer scope.

## When Should I Use var Instead of let or const?

**NEVER!**

You might be wondering:

"Why wouldn't I always use global variables? Then, I would never need to use function arguments since ALL my functions would have access to EVERYTHING!"

Well... Global variables might seem like a convenient idea at first, especially when you're writing small scripts and programs, but there are many reasons why you shouldn't use them unless you have to. For instance, global variables can conflict with other global variables of the same name. Once your programs get larger and larger, it'll get harder and harder to keep track and prevent this from happening.

There are also other reasons you'll learn more about in more advanced courses. But for now, just work on minimizing the use of global variables as much as possible.

💡 Always use let and const instead of var.

# Scope Gotcha I: Shadowing

## Scope Can Be Tricky!

Scope can be a tricky subject, especially when you're working in both global and function scope.

**Shadowing** occurs when variables in different scopes have the same name. When this happens the variable in the inner scope overrides the variable in the outer scope.

***Example: scope shadowing***

**let** bookTitle = "Le Petit Prince";

**console**.**log**(bookTitle);

**function** **displayBookEnglish**() {

bookTitle = 'The Little Prince';

**console**.**log**(bookTitle);

}

**displayBookEnglish**()

**console**.**log**(bookTitle);

**Prints**

Le Petit Prince  
The Little Prince  
The Little Prince

### Best Practice: Declare a New Variable

To avoid scope override, always declare a new variable inside your function. This prevents JavaScript from reassigning the value of the variable in the outer scope.

***Example: no shadowing***

**let** bookTitle = "Le Petit Prince";

**console**.**log**(bookTitle);

**function** **displayBookEnglish**() {

**let** bookTitle = 'The Little Prince';

**console**.**log**(bookTitle);

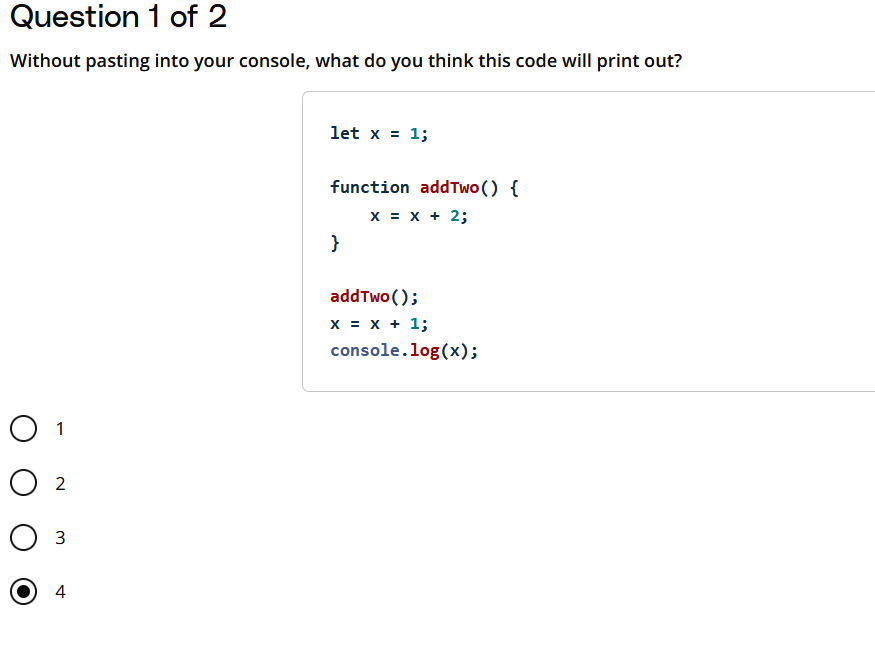
}

**displayBookEnglish**()

**console**.**log**(bookTitle);

**Prints**

Le Petit Prince  
The Little Prince  
Le Petit Prince



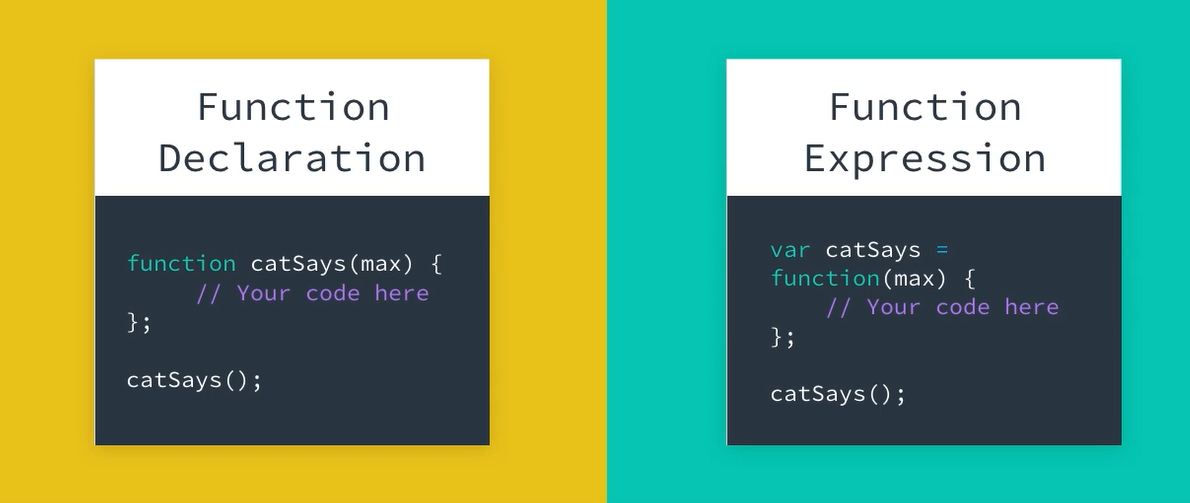
**You may be wondering...** Why are we using *var* in the second quiz?  
Great question! There are some interesting implications of block scoping that we will cover later in the lesson. If you want a preview, try replacing var with let and see what happens when you run the code. We'll explore this more when we look into hoisting.

# Scope Recap

## What You’ve Learned So Far:

* If an identifier is declared in **global scope**, it's available everywhere.
* If an identifier is declared in **function scope**, it's available in the function it was declared in (even in functions declared inside the function).
* If an identifier is declared in **block scope** with var, it is available in the block and in the outer scope of the block it was declared in.
* If an identifier is declared in **block scope** with let or const, it is only available in the block it was declared in.
* When trying to access an identifier, the JavaScript Engine will first look in the current function. If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there. It will keep doing this until it reaches the global scope.
* Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

# Function Expressions



Once you know how to declare a function, a whole new set of possibilities will open up to you.

For instance, remember how you can store anything you want in a variable? Well, in JavaScript, you can also store functions in variables. When a function is stored inside a variable it's called a **function expression**.

**const** **catSays** = **function**(max) {

**let** catMessage = "";

**for** (**let** i = 0; i < max; i++) {

catMessage += "meow ";

}

**return** catMessage;

};

Notice how the function keyword no longer has a name.

**const** **catSays** = **function**(max) {

*// code here*

};

It's an **anonymous function**, a function with no name, and you've stored it in a variable called catSays.

And, if you try accessing the value of the variable catSays, you'll even see the function returned back to you.

catSays;

**Returns:**

**function**(max) {

**let** catMessage = ""

**for** (**let** i = 0; i < max; i++) {

catMessage += "meow ";

}

**return** catMessage;

}

## Function Expressions vs Function Declarations

Deciding when to use a function expression and when to use a function declaration can depend on a few things, and you will see some ways to use them in the next section. But, one thing you'll want to be careful of is hoisting. We'll look at that next.

# Scope Gotcha II: Hoisting

## Hoisting

Sometimes your JavaScript code will produce errors that may seem counterintuitive at first. **Hoisting** is another one of those topics that might be the cause of some of these tricky errors you're debugging.

Hoisting is a result of how JavaScript is interpreted by your browser. Essentially, before any JavaScript code is executed, all function declarations and variables declared with var are **hoisted**, which means they're raised to the top of the function scope.

Let's take a look at an example:

## JavaScript is Different!

In most programming languages, you have to declare a function or variable before you can call it.

Intuitively, you might think **"This code shouldn't work!"** because we're trying to call findAverage before it is declared:

**findAverage**(5, 9);

**function** **findAverage**(x, y) {

**var** answer = (x + y) / 2;

**return** answer;

}

But, surprisingly it does work!

**Returns** 7

in JavaScript we can call a function before we declare it due to **hoisting**. Before any JavaScript code is executed, all function declarations and variables declared with var are hoisted to the top of their current scope.

So even though the code you write doesn't change, the JavaScript engine interprets it as if you wrote it this way:

**function** **findAverage**(x, y) {

**var** answer = (x + y) / 2;

**return** answer;

}

**findAverage**(5, 9);

### Demo Code

Hosting can lead to odd results.

As expected, this code returns a reference error because we haven't defined greeting anywhere:

**function** **sayGreeting**() {

**console**.**log**(greeting);

}

**sayGreeting**()

**Returns** Uncaught ReferenceError: greeting is not defined

To get rid of the error, we can define greeting anywhere inside the sayGreeting function, even after we call it:

**function** **sayGreeting**() {

**console**.**log**(greeting);

**var** greeting;

}

**sayGreeting**()

**Returns** undefined

We don't get an error, but we do get undefined. Can we fix this by assigning a value to greeting when we declare it? No. This code still returns undefined:

**function** **sayGreeting**() {

**console**.**log**(greeting);

**var** greeting = "hello";

}

**sayGreeting**()

**Returns** undefined

This is because with hoisting, the variable declaration is being hoisted to the top of the function, but the value of greeting isn't assigned until after the console.log statement is run. JavaScript is interpreting the code as if it were written like this:

**function** **sayGreeting**() {

**var** greeting;

**console**.**log**(greeting);

greeting = "hello";

}

**sayGreeting**()

**Returns** undefined

To avoid bugs like this, declare your functions at the top of your scripts and declare and assign your variables at the top of your functions.

**function** **sayGreeting**() {

**var** greeting = "hello";

**console**.**log**(greeting);

greeting

}

**sayGreeting**()

**Prints** Hello

# More About Hoisting

## What About Function Expressions?

***Function expressions are not hoisted***, since they involve variable assignment, and only variable declarations are hoisted. The function expression will not be loaded until the interpreter reaches it in the script.

### Example: Function Expressions vs Function Declarations

The function expression meow is not hoisted so this code throws an error:

**function** **cat**() {

**console**.**log**(**meow**(2));

**const** **meow** = **function** (max) {

**let** catMessage = '';

**for** (**let** i = 0; i < max; i++) {

catMessage = 'meow ';

}

**return** catMessage;

};

**function** **purr**() {

**return** 'purrrr!';

}

}

**cat**();

**Returns** Uncaught ReferenceError: Cannot access 'meow' before initialization

The function declaration purr is hoisted so this code runs without error:

**function** **cat**() {

**console**.**log**(**purr**());

**const** **meow** = **function** (max) {

**let** catMessage = '';

**for** (**let** i = 0; i < max; i++) {

catMessage = 'meow ';

}

**return** catMessage;

};

**function** **purr**() {

**return** 'purrrr!';

}

}

**cat**();

**Prints** purrrr!

## Hoisting Is Another Reason to Use let and const Instead of var!

Variables declared with let and const eliminate the issue of variable hoisting because they’re scoped **to the block**, not to the function.

When a variable is declared using let or const inside a block of code (denoted by curly braces { }), the variable is stuck in the **temporal dead zone** until the variable’s declaration is processed. This sounds scary, but it basically means that the code cannot access the variable before it has been declared. If you try, you'll get a Reference Error

# Hoisting Recap

## What You’ve Learned So Far:

* JavaScript hoists function declarations and variables declared with var to the top of the current scope.
* Hoisting doesn't happen when variables are declared with let or const.
* Variable assignments are not hoisted so function expressions are not hoisted.

### Best Practices

* Declare functions and variables at the top of your scripts, so the syntax and behavior are consistent with each other.
* Use let and const to declare variables. You may see var in legacy code, but do not use it in any new code you are writing.

# Quiz: Build a Triangle

## Directions:

For this quiz, you're going to create a function called buildTriangle() that will accept an input (the triangle at its widest width) and will return the string representation of a triangle. See the example output below.

**buildTriangle**(10);

**Returns**:

\*

\* \*

\*\*\*

\*\*\* \*

\*\*\* \* \*

\*\*\* \*\*\*

\*\*\* \*\*\* \*

\*\*\* \*\*\* \* \*

\*\*\* \*\*\* \*\*\*

\* \* \* \* \* \* \* \* \* \*

We've given you one function makeLine() to start with. The function takes in a line length, and builds a line of asterisks and returns the line with a newline character.

**function** **makeLine**(length) {

**var** **let** = "";

**for** (**let** j = 1; j <= length; j++) {

line += "\* "

}

**return** line + "\n";

}

**You will need to call this makeLine() function in buildTriangle().**

### Think It Through!

This will be the most complicated program you've written yet, so take some time thinking through the problem before diving into the code. What tools will you need from your JavaScript tool belt? Professionals plan out their code before writing anything. Think through the steps your code will need to take and write them down in order. Then go through your list and convert each step into actual code. Good luck!

### Running Your Code

Enter the following in the terminal:

**node** triangle.js

### Testing Your Code

Change the parameter in the testing code to confirm that your program works for different values of length.

For console.log(buildTriangle(3)); the output should be:

*\**

*\** *\**

*\**\*\*

If you try with console.log(buildTriangle(6)); the following should be logged into the console:

*\**

*\** *\**

*\**\**\**

*\**\*\* \*

\*\*\* \* \*

\*\*\* \*\*\*

Lastly, console.log(buildTriangle(10)); should log out:

*\**

*\** *\**

*\**\**\**

*\**\*\* \*

\*\*\* \* \*

\*\*\* \*\*\*

\*\*\* \*\*\* \*

\*\*\* \*\*\* \* \*

\* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

// Example usage

console.log(laugh(3)); // Output: "hahaha!"

// creates a line of ,[object Object], ";

}

return line + "\n";

}

// your code goes here.  Make sure you call makeLine() in your own code.

function buildTriangle(length) {

// Let's build a huge string equivalent to the triangle

var triangle = "";

//Let's start from the topmost line

let lineNumber = 1;

for(lineNumber=1; lineNumber<=length; lineNumber++){

    // We will not print one line at a time.

    // Rather, we will make a huge string that will comprise the whole triangle

    triangle = triangle + makeLine(lineNumber);

}

return triangle;

}

// test your code

console.log(buildTriangle(10));

// Provided makeLine function with a typo corrected

function makeLine(length) {

  let line = "";  // Corrected typo here from `var let` to `let line`

  for (let j = 1; j <= length; j++) {

    line += "\* ";

  }

  return line + "\n";

}

// Build the buildTriangle function

function buildTriangle(width) {

  let triangle = "";  // Initialize an empty string to build the triangle

  for (let i = 1; i <= width; i++) {  // Loop from 1 up to the width

    triangle += makeLine(i);  // Add each line to the triangle

  }

  return triangle;  // Return the complete triangle

}

// Call the function and print the result

console.log(buildTriangle(10));

# Patterns with Function Expressions

## Functions as Parameters

Being able to store a function in a variable makes it really simple to pass the function into another function. A function that is passed into another function is called a **callback**. Let's say you had a helloCat() function, and you wanted it to return "Hello" followed by a string of "meows" like you had with catSays. Well, rather than redoing all of your hard work, you can make helloCat() accept a callback function, and pass in catSays.

*// function expression catSays*

**const** **catSays** = **function**(max) {

**let** catMessage = "";

**for** (**let** i = 0; i < max; i++) {

catMessage += "meow ";

}

**return** catMessage;

};

*// function declaration helloCat accepting a callback*

**function** **helloCat**(callbackFunc) {

**return** "Hello " + **callbackFunc**(3);

}

*// pass in catSays as a callback function*

**helloCat**(catSays);

**Returns** 'Hello meow meow meow '

You can create a function expression with a named function, like this:

**const** **favoriteMovie** = **function** **movie**() {

**return** "The Fountain";

};

But you still need to use its assigned identifier to call it:

**favoriteMovie**();

**Returns** 'The Fountain'

If you try to use the function name to call the function you'll get an error:

**movie**();

**Returns** Uncaught ReferenceError: movie is not defined

## Inline Function Expressions

A function expression is when a function is assigned to a variable. And, in JavaScript, this can also happen when you pass a function inline as an argument to another function. Take the favoriteMovie example for instance:

*// Function expression that assigns the function displayFavorite*

*// to the variable favoriteMovie*

**const** **favoriteMovie** = **function** **displayFavorite**(movieName) {

**console**.**log**("My favorite movie is " + movieName);

};

*// Function declaration that has two parameters: a function for displaying*

*// a message, along with a name of a movie*

**function** **movies**(messageFunction, name) {

**messageFunction**(name);

}

*// Call the movies function, pass in the favoriteMovie function and name of movie*

**movies**(favoriteMovie, "Finding Nemo");

**Returns:** My favorite movie is Finding Nemo

But you could have bypassed the first assignment of the function, by passing the function to the movies() function inline.

*// Function declaration that takes in two arguments: a function for displaying*

*// a message, along with a name of a movie*

**function** **movies**(messageFunction, name) {

**messageFunction**(name);

}

*// Call the movies function, pass in the function and name of movie*

**movies**(**function** **displayFavorite**(movieName) {

**console**.**log**("My favorite movie is " + movieName);

}, "Finding Nemo");

**Returns:** My favorite movie is Finding Nemo

This type of syntax, writing function expressions that pass a function into another function inline, is really common in JavaScript. It can be a little tricky at first, but be patient, keep practicing, and you'll start to get the hang of it!

### Why Use Anonymous Inline Function Expressions?

Using an anonymous inline function expression might not seem useful at first. Why define a function that can only be used once and you can't even call it by name?

Anonymous inline function expressions are often used with function callbacks that are unlikely to be reused elsewhere. Yes, you could store the function in a variable, give it a name, and pass it in like you saw in the examples above. However, when you know the function is not going to be reused, it could save you many lines of code to just define it inline.

For example, we can make the callback function in movies an anonymous inline function like this:

**function** **movies**(messageFunction, name) {

**messageFunction**(name);

}

**movies**(**function** (movieName) {

**console**.**log**("My favorite movie is " + movieName);

}, "Finding Nemo");

# Function Expression Recap

## What You’ve Learned So Far:

**Function Expression**: When a function is assigned to a variable. The function can be named, or anonymous. Use the variable name to call a function defined in a function expression.

*// anonymous function expression*

**const** **doSomething** = **function**(y) {

**return** y + 1;

};

*// named function expression*

**const** **doSomething** = **function** **addOne**(y) {

**return** y + 1;

};

*// for either of the definitions above, call the function like this:*

**doSomething**(5);

**Returns:** 6

You can even pass a function into another function inline. This pattern is commonly used in JavaScript, and can be helpful streamlining your code.

*// function declaration that takes in two arguments: a function for displaying*

*// a message, along with a name of a movie*

**function** **movies**(messageFunction, name) {

**messageFunction**(name);

}

*// call the movies function, pass in the function and name of movie*

**movies**(**function** **displayFavorite**(movieName) {

**console**.**log**("My favorite movie is " + movieName);

}, "Finding Nemo");

# Quiz: Laugh

## Directions:

Write an anonymous function expression that:

* stores a function in a variable called "laugh"
* creates a string with the number of "ha"s that you pass in as an argument
* adds an exclamation point at the end of the string
* returns the string

**laugh**(3);

**Returns:** hahaha!

// Call the function and print the result

console.log(buildTriangle(10));

// Define an anonymous function and assign it to the variable laugh

const laugh = function(num) {

  let sound = "";  // Initialize an empty string

  for (let i = 0; i < num; i++) {

      sound += "ha";  // Append "ha" for each count up to num

  }

  return sound + "!";  // Add exclamation mark at the end

};

// Test the function with laugh(3)

console.log(laugh(3));  // Expected output: "hahaha!"

# Quiz: Cry

## Directions:

Write a named function expression that stores the function in a variable called cry and returns "boohoo!". Don't forget to call the function using the variable name, not the function name:

**cry**();

const cry = function myFunction(){

  const sound = "boohoo!";

  return sound;

};

// test your solution

console.log(cry());

# Quiz: Inline

## Directions:

Call the emotions() function so that it prints the output you see below, but instead of passing the laugh() function as an argument, pass an inline function expression instead.

**emotions**("happy", **laugh**(2)); *// you can use your laugh function from the previous quizzes*

**Prints:** "I am happy, haha!

### Testing Your Code

How will you know if your code works? Change the value of the argument in the myFunc call in the emotions function, re-run the code and check the output:

| **myFunc argument** | **Expected Output** |
| --- | --- |
| 0 | I am happy, ! |
| 2 | I am happy, haha! |
| 7 | I am happy, hahahahahahaha! |

// Call the emotions() function with two arguments

// Argument 1 - "happy" string

// Argument 2 - an inline function expression

emotions ("happy", function(num) {

  let sound = ""; // Local variable

  //Iterate

  for (let i = 0 ; i < num ; i++) {

      sound = sound + "ha" ;

  }

  sound = sound +"!";

  return sound;

});

# Lesson Summary

## Functions are Fundamental to Programming!

Congrats on learning how to use functions. We covered a lot in this lesson. You now know how to:

* Declare and run functions
* Use return statements to return items when a function runs
* Explain the difference between parameters and arguments
* Explain the difference between returning and logging
* Understand how scope works in JavaScript
* Use let and const in block scope
* Understand how hosting works
* Store functions in variables using function expressions
* Use inline function expressions

Great work!

# Intro to Arrays

## Arrays Allow Us to Store Lists of Data

In this lesson we'll learn about:

* How to create arrays
* How to use arrays
* How arrays are structured

And we'll be talking a lot about donuts and how to keep track of all of the donuts in our shop!

# Donuts to Code using array



## Hurray for Arrays!

* An array is a data structure we can use to store **multiple values**.
* Values in an array are **ordered** (like a numbered list).
* The numbering of the array order starts at 0 -- not at 1

## Building Our Donuts Array

We started with:

**const** donut1 = "glazed";

**const** donut2 = "chocolate frosted";

**const** donut3 = "cinnamon";

**const** donut4 = "sprinkled";

**const** donut5 = "powdered";

**const** donut6 = "cinnamon sugar";

**const** donut7 = "glazed cruller";

**const** donut8 = "chocolate cruller";

**const** donut9 = "cookies";

**const** donut10 = "Boston creme";

**const** donut11 = "powdered jelly filled";

**const** donut12 = "creme de leche";

**const** donut13 = "glazed donut holes";

**const** donut14 = "blueberry donut holes";

**const** donut15 = "cake donut holes";

**const** donut16 = "chocolate donut holes";

And we created an array like this:

**const** donuts = [ "glazed", "chocolate frosted", "cinnamon", "sprinkled", "powdered", "cinnamon sugar", "glazed cruller", "chocolate cruller", "cookies", "Boston creme", "powdered jelly filled", "creme de leche", "glazed donut holes", "blueberry donut holes", "cake donut holes", "chocolate donut holes" ];

Sometimes it is easier to view an array with one item on each line, like this:

**const** donuts = [ *// start of the array*

"glazed", *// each element separated by a comma*

"chocolate frosted",

"cinnamon",

"sprinkled",

"powdered",

"cinnamon sugar",

"glazed cruller",

"chocolate cruller",

"cookies",

"Boston creme",

"powdered jelly filled",

"creme de leche",

"glazed donut holes",

"blueberry donut holes",

"cake donut holes",

"chocolate donut holes"

]; *// end of the array*

Either way is valid!

***Keyboard Shortcut Magic*** 🪄

James uses a lot of keyboard shortcuts in this video! The keyboard shortcuts are specific to the editor, as well as the platform you are using. Every editor has its own set of keyboard shortcuts bound to a particular language. For example, if you are using Visual Studio Code or Sublime Text as an editor, you can find the default shortcuts within the editor's menu options.\*\*\*  
\*\*\*Have fun exploring your code editor's shortcuts!

### A Question you may be asking... 🤔

**Why does the video use** var **and the example code use** const**?**

Great question! While you can use var to declare an array, it is now considered best practice to use let or const instead. We're showing you both but you should try to use let and const in your own code.

We'll learn more about why we use const after we learn how to access array elements using the index.

# Creating an Array

## Arrays

An **array** is useful because it stores multiple values into a single, organized data structure. You can define a new array by listing values separated with commas between square brackets [].

*// creates a new empty array*

**const** emptyArray = [];

*// creates a `donuts` array with three strings*

**const** donuts = ["glazed", "powdered", "jelly"];

But strings aren’t the only type of data you can store in an array. You can also store numbers, booleans… and really anything!

*// creates a `mixedData` array with mixed data types*

**const** mixedData = ["abcd", 1, true, **undefined**, **null**, "all the things"];

You can even store an array in an array to create a **nested array**!

*// creates a `arraysInArrays` array with three arrays*

**const** arraysInArrays = [[1, 2, 3], ["Julia", "James"], [true, false, true, false]];

Nested arrays can be particularly hard to read, so it's common to write each nested array on one line, using a newline after each comma:

**const** arraysInArrays = [

[1, 2, 3],

["Julia", "James"],

[true, false, true, false]

];

Later in this lesson, we’ll look into some unique situations where nested arrays can be useful.

### Using the Array Constructor

You may also see arrays created this way:

*// creates a new empty array*

**const** emptyArray = **new** **Array**();

*// creates a `donuts` array with three strings*

**const** donuts = **new** **Array**("glazed", "powdered", "jelly");

This syntax is valid, but it is preferred to use the literal constructor because the syntax is simpler and more intuitive.

*// creates a new empty array*

**const** emptyArray = [];

*// creates a `donuts` array with three strings*

**const** donuts = ["glazed", "powdered", "jelly"];

# Accessing Array Elements

## Elements and Indexes

* An **element** is an individual piece of data in an array.
* Each element in the array is numbered, starting with 0. This number is called the **index**.
* The index allows us to access the element's position in the array.

**const** donuts = [

"glazed", *// index is 0*

"chocolate frosted", *// index is 1*

"cinnamon", *// index is 2*

"sprinkled", *// index is 3*

"powdered", *// index is 4*

"cinnamon sugar", *// index is 5*

"glazed cruller", *// index is 6*

"chocolate cruller", *// index is 7*

"cookies", *// index is 8*

"Boston creme", *// index is 9*

"powdered jelly filled", *// index is 10*

"creme de leche", *// index is 11*

"glazed donut holes", *// index is 12*

"blueberry donut holes", *// index is 13*

"cake donut holes", *// index is 14*

"chocolate donut holes" *// index is 15*

];

**console**.**log**(donuts[11]); *// "creme de leche"*

# Array Index

## Using the Array Index to Access an Element

Remember that elements in an array are indexed starting at position 0. To access an element in an array, use the name of the array immediately followed by square brackets containing the index of the value you want to access.

**const** donuts = ["glazed", "powdered", "sprinkled"];

**console**.**log**(donuts[0]); *// "glazed" is the first element in the `donuts` array*

**Prints:** "glazed"

### Accessing a Non-existant Element

One thing to be aware of is if you try to access an element at an index that does not exist, a value of undefined will be returned.

**console**.**log**(donuts[3]); *// the fourth element in the `donuts` array does not exist!*

**Prints:** undefined

### Changing the Value of an Array Element

If you want to change the value of an element in array, you can do so by setting it equal to a new value.

donuts[1] = "glazed cruller"; *// changes the second element in the `donuts` array to "glazed cruller"*

**console**.**log**(donuts[1]);

**Prints:** "glazed cruller"

#### **Why are we using** const **instead of** let **when we declare arrays?**

Another great question!

A simple way to think about the difference between let and const is that we use let when we anticipate that the value of a variable will change and const when we think it will be constant -- but that shorthand doesn't tell the whole story. The difference between let and const is not so much about ***change*** but about ***reassignment***

* let allows you to reassign the variable
* const doesn't allow you to reassign the variable

The decision about whether we need to be able to reassign the variable is based on what type of variable it is and how JavaScript stores those values.

#### Primitive vs. Object Types in JavaScript

***String***, ***Number***, ***Boolean***, ***undefined*** and ***null*** are considered [***Primitive Types***(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_values) in JavaScript. These relatively simple data types represent just one value which makes it easy for JavaScript to store that value. So when you assign a primitive value to a variable, JavaScript actually assigns that value.

***Arrays*** are more complicated because they consist of a list of values which makes storage much more complicated. Arrays are actually [***Object types***(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects) which means that instead of assigning all of the values of the list to the array, JavaScript simply assigns a reference to where to find the values in the array. Even if the values inside the object change, the reference address doesn't.

Here's an analogy that might help. Think of a JavaScript array as if it were a house. The house has a group of people who live inside it. If those people move out, and a new group of people moves in, the names of the people inside the house changes, but the house's postal address won't.

### Test Your Intuition About let and const

### TLDR;

We use const to declare arrays because JavaScript is assigning a reference that points to that array. We can change whatever we want inside the array, but we cannot change which array the variable points to.

# Quiz: UdaciFamily

## Directions:

Create an array called udaciFamily and add "Julia", "James", and your name to the array. Then, print the udaciFamily to the console using console.log.

### Running Your Code

Enter the following in the terminal:

// Solution 1

// Create array using array literal

const udaciFamily = ["Julia", "James", "Yourname"];

// Iterate over the array elements

// We can find the size of an array using its <code>length</code> property

// Remember that the indexing of elements in an array starts from 0

for (let index = 0; index < udaciFamily.length; index++){

    console.log(udaciFamily[index]);

}

// Solution 2

// Create array using the Array constructor

const udaciFamily = new Array("Julia", "James", "Yourname");

// Iterate over the array elements

for (let index = 0; index < udaciFamily.length; index++){

    console.log(udaciFamily[index]);

}

// Solution 1: Using Array Literal

const udaciFamily1 = ["Julia", "James", "Wagari"];

for (let index = 0; index < udaciFamily1.length; index++){

    console.log(udaciFamily1[index]);

}

// Solution 2: Using Array Constructor

const udaciFamily2 = new Array("Julia", "James", "Wagari");

for (let index = 0; index < udaciFamily2.length; index++){

    console.log(udaciFamily2[index]);

}

# Quiz: Building the Crew

The space western TV drama [Firefly(opens in a new tab)](https://en.wikipedia.org/wiki/Firefly_%28TV_series%29) premiered in the United States on September 20, 2002. Although the show only featured fourteen episodes and was canceled during its first season, it continues to remain popular today with a growing fan base. In the show, the captain [Mal(opens in a new tab)](https://en.wikipedia.org/wiki/Malcolm_Reynolds), played by [Nathan Fillion(opens in a new tab)](https://en.wikipedia.org/wiki/Nathan_Fillion), leads a crew of renegades on the spaceship Serenity.

## Directions:

Create an array called crew to organize the Serenity’s crew and set it equal to the variables below . You don't need to type out the actual strings, just use the provided variables.

**const** captain = "Mal";

**const** second = "Zoe";

**const** pilot = "Wash";

**const** companion = "Inara";

**const** mercenary = "Jayne";

**const** mechanic = "Kaylee";

Then, print the crew array to the console.

//Solution 1 - Array literal

const crew = [captain, second, pilot, companion, mercenary, mechanic];

// Print

console.log(crew);

//Solution 2 - Array constructor

const crew = new Array(captain, second, pilot, companion, mercenary, mechanic);

// Print

console.log(crew);

# Quiz: The Price is Right

## Directions:

Starting with this array of prices, double the prices of the 1st, 3rd, and 7th elements in the array.

**const** prices = [1.23, 48.11, 90.11, 8.50, 9.99, 1.00, 1.10, 67.00];

**TIP:** The 1st element of any array has an index of 0.

Afterwards, print out the prices array to the console.

const prices = [1.23, 48.11, 90.11, 8.50, 9.99, 1.00, 1.10, 67.00];

for (let index = 0; index < prices.length; index++){

  //change the value of first element

  if (index === 0) {

    prices[index] = prices[index] \* 2;

  }

  //change the value of third element

  else if (index === 2) {

    prices[index] = prices[index] \* 2;

  }

  //change the value of seventh element

  else if (index === 6) {

    prices[index] = prices[index] \* 2;

  }

}

console.log(prices);

# Array Properties and Methods

[***https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array***](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

## Arrays are Powerful Data Structures

Arrays have a number of properties and built-in methods.

Array properties includelengthwhich is similar to the length method for String types.

Common Array methods include:

* reverse: reverses the order of the elements in an array
* sort: sorts the elements in an array
* push: adds elements to an array
* pop: removes elements from an array

To see all of the properties and built-in methods for modifying arrays and accessing values in an array, check out the [MDN Documentation(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

***TIP***: You can type []. into the JavaScript console for a list of all the available Array methods.

# Length

## Array.length

You can find the **length** of an array by using its length property.

**const** donuts = ["glazed", "powdered", "sprinkled"];

**console**.**log**(donuts.length);

**Prints:** 3

To access the length property, type the name of the array, followed by a period . (you’ll also use the period to access other properties and methods), and the word length. The length property will then return the **number of elements** in the array.

**TIP:** Strings have a length property too! You can use it to get the length of any string. For example, "supercalifragilisticexpialidocious".length returns 34.

# Push

o you can find length of an array, but what if you want to modify an array?

Thankfully, arrays have quite a few built-in methods for adding and removing elements from an array. The two most common methods for modifying an array are push() and pop().

## Push

You can use the push() method to add elements to the end of an array.

For example, imagine the following spread of donuts.



You can represent the spread of donuts using an array.

**const** donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled"];```

# Pop

## Pop

Alternatively, you can use the pop() method to remove elements from the end of an array.

**const** donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled", "powdered"];

donuts.**pop**(); *// pops "powdered" off the end of the `donuts` array*

donuts.**pop**(); *// pops "sprinkled" off the end of the `donuts` array*

donuts.**pop**(); *// pops "cinnamon sugar" off the end of the `donuts` array*

**Returns:** "cinnamon sugar"  
**donuts array:** ["glazed", "chocolate frosted", "Boston creme", "glazed cruller"]

With the pop() method you don’t need to pass a value; instead, pop() will always remove the last element from the end of the array.

Notice that pop() returns the element that has been removed. This can be handy if you want to do something with the element that you removed from the array.

**const** donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled", "powdered"];

donuts.**pop**(); *// the `pop()` method returns "powdered" because "powdered" was the last element on the end of `donuts` array*

**Returns:** "powdered"

# Splice

## Splice

splice() is another handy method that allows you to add and remove elements from anywhere within an array.

While push() and pop() limit you to adding and removing elements from the end of an array, splice() lets you specify the index location to add new elements, as well as the number of elements you'd like to delete (if any).

**const** donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller"];

donuts.**splice**(1, 1, "chocolate cruller", "creme de leche"); *// removes "chocolate frosted" at index 1 and adds "chocolate cruller" and "creme de leche" starting at index 1*

**Returns**: ["chocolate frosted"]  
**donuts** array after calling the splice() method: ["glazed", "chocolate cruller", "creme de leche", "Boston creme", "glazed cruller"]

**Following is the syntax of** **splice()** **method**: arrayName.splice(arg1, arg2, item1, ....., itemX); where,

* arg1 = Mandatory argument. Specifies the starting index position to add/remove items. You can use a negative value to specify the position from the end of the array e.g., -1 specifies the last element.
* arg2 = Optional argument. Specifies the count of elements to be removed. If set to 0, no items will be removed.
* item1, ....., itemX are the items to be added at index position arg1

splice() method returns the item(s) that were removed.

Note - This video does not have an audio. It was created as a visual to aid learning.

splice() is an incredibly powerful method that allows you to manipulate your arrays in a variety of ways. Any combination of adding or removing elements from an array can all be done in one simple line of code.

Take a look at the [MDN documentation(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) to see a long list of example code snippets demonstrating the power of splice() and then try the next set of programming quizzes.

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



# Quiz: Colors of the Rainbow

## Directions:

James was creating an array with the colors of the rainbow, and he forgot some colors. The standard rainbow colors are usually listed in this order:

**const** rainbow = ["Red", "Orange", "Yellow", "Green", "Blue", "Purple"];

but James had this:

**const** rainbow = ["Red", "Orange", "Blackberry", "Blue"];

Using only the splice() method, insert the missing colors into the array, and remove the color "Blackberry" by following these steps:

1. Remove "Blackberry"
2. Add "Yellow" and "Green"
3. Add "Purple"

const rainbow = ["Red", "Orange", "Blackberry", "Blue"];

// Step 1: Remove "Blackberry" and add "Yellow" and "Green" in its place.

rainbow.splice(2, 1, "Yellow", "Green");

// Step 2: Add "Purple" at the end of the array.

rainbow.splice(5, 0, "Purple");

console.log(rainbow);

# Quiz: Quidditch Cup

In the [Harry Potter(opens in a new tab)](https://en.wikipedia.org/wiki/Harry_Potter) novels, children attending the [Hogwarts School of Witchcraft and Wizardry(opens in a new tab)](https://en.wikipedia.org/wiki/Hogwarts) belong to one of four houses: Gryffindor, Hufflepuff, Ravenclaw, or Slytherin. Each year, the houses assemble a [Quidditch(opens in a new tab)](https://en.wikipedia.org/wiki/Quidditch) team of seven players to compete for the coveted Quidditch Cup. Your task is to help the coach determine if the team has enough players.

## Directions:

Create a function called hasEnoughPlayers() that takes the team array as an argument and returns true or false depending on if the array has at least seven players.

Test your code with the following teams:

**const** team1 = ["Oliver Wood", "Angelina Johnson", "Katie Bell", "Alicia Spinnet", "George Weasley", "Fred Weasley", "Harry Potter"];

**const** team2 = ["George Weasley", "Fred Weasley", "Harry Potter"];

**const** team3 = [];

**const** team4 = ["Oliver Wood", "Angelina Johnson", "Katie Bell", "Ali

function hasEnoughPlayers(arrayInstance) {

  if (arrayInstance.length >= 7) {

      return true;

  } else {

      return false;

  }

}

const team1 = ["Oliver Wood", "Angelina Johnson", "Katie Bell", "Alicia Spinnet", "George Weasley", "Fred Weasley", "Harry Potter"];

const team2 = ["George Weasley", "Fred Weasley", "Harry Potter"];

const team3 = [];

const team4 = ["Oliver Wood", "Angelina Johnson", "Katie Bell", "Alicia Spinnet", "George Weasley", "Fred Weasley", "Harry Potter", "Hermione Granger", "Ron Weasley", "Neville Longbottom"];

console.log(hasEnoughPlayers(team1)); // Expected output: true

console.log(hasEnoughPlayers(team2)); // Expected output: false

console.log(hasEnoughPlayers(team3)); // Expected output: false

console.log(hasEnoughPlayers(team4)); // Expected output: true

# Quiz: Joining the Crew

## Directions:

In an earlier exercise, you created a crew array to represent Mal’s crew from the hit show Firefly.

**const** captain = "Mal";

**const** second = "Zoe";

**const** pilot = "Wash";

**const** companion = "Inara";

**const** mercenary = "Jayne";

**const** mechanic = "Kaylee";

**const** crew = [captain, second, pilot, companion, mercenary, mechanic];

Later in the show, Mal takes on three new crew members named "Simon", "River", and "Book". Use the push() method to add the three new crew members to the crew array.

**const** doctor = "Simon";

**const** sister = "River";

**const** shepherd = "Book";

// Existing crew members

const captain = "Mal";

const second = "Zoe";

const pilot = "Wash";

const companion = "Inara";

const mercenary = "Jayne";

const mechanic = "Kaylee";

// Create the initial crew array

const crew = [captain, second, pilot, companion, mercenary, mechanic];

// New crew members to be added

const doctor = "Simon";

const sister = "River";

const shepherd = "Book";

// Use the push() method to add new crew members to the crew array

crew.push(doctor);

crew.push(sister);

crew.push(shepherd);

// Output the updated crew array

console.log(crew);

# Array Loops

## Array Loops

Once the data is in the array, you want to be able to efficiently access and manipulate each element in the array without writing repetitive code for each element.

For instance, if this was our original donuts array:

**const** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

and we decided to make all the same donut types, but only sell them as donut holes instead, we could write the following code:

donuts[0] += " hole";

donuts[1] += " hole";

donuts[2] += " hole";

**donuts array:** ["jelly donut hole", "chocolate donut hole", "glazed donut hole"]

But remember, you have another powerful tool at your disposal, **loops**!

To loop through an array, you can use a variable to represent the index in the array and then loop over that index to perform whatever manipulations your heart desires.

**const** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

*// the variable `i` is used to step through each element in the array*

**for** (**let** i = 0; i < donuts.length; i++) {

donuts[i] += " hole";

donuts[i] = donuts[i].**toUpperCase**();

}

**donuts array:** ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]

In this example, the variable i is being used to represent the index of the array. As i is incremented, you are stepping over each element in the array starting from 0 until donuts.length - 1 (donuts.length is out of bounds).

# The forEach Loop

## The forEach() loop

### for Loop Review

The for loop should be familiar to you:

**const** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

**for** (**let** i = 0; i < donuts.length; i++ ) {

donuts[i] += " hole";

donuts[i] = donuts[i].**toUpperCase**();

}

**console**.**log**(donuts); *// ['JELLY DONUT HOLE', 'CHOCOLATE DONUT HOLE', 'GLAZED DONUT HOLE'*

## Array Methods for Looping

Arrays have a set of special methods to help you iterate over and perform operations on collections of data. You can view the [MDN Documentation list of Array methods(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), but a couple big ones to know are the forEach() and map() methods.

### The forEach Method

The forEach() method gives you an alternative way to iterate over an array, and manipulate each element in the array with an inline function expression.

**const** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

donuts.**forEach**(**function**(donut) {

donut += " hole";

donut = donut.**toUpperCase**();

**console**.**log**(donut);

});

**Prints:**

JELLY DONUT HOLE

CHOCOLATE DONUT HOLE

GLAZED DONUT HOLE

Notice that the forEach() method iterates over the array without the need of an explicitly defined index. In the example above, donut corresponds to the element in the array itself. This is different from a for or while loop where an index is used to access each element in the array:

**for** (**let** i = 0; i < donuts.length; i++) {

donuts[i] += " hole";

donuts[i] = donuts[i].**toUpperCase**();

**console**.**log**(donuts[i]);

}

### forEach Syntax

The function that you pass to the forEach() method can take up to three parameters. We have named them element, index, and array, but you can name them whatever you like.

**const** myArray = ["A", "B", "C", "D"];

**function** **myAwesomeFunction**(element, index, array) {

**console**.**log**("Element: ", element);

**console**.**log**("Index: ", index);

**console**.**log**("Array: ", array);

}

myArray.**forEach**(myAwesomeFunction);

The forEach() method will call this function once for each element in the array (hence the name forEach.) Each time, it will call the function with different arguments. The element parameter will get the value of the array element. The index parameter will get the index of the element (starting with zero). The array parameter will get a reference to the whole array, which is handy if you want to modify the elements.

HHere's another example:

words = ["cat", "in", "hat"];

words.**forEach**(**function**(word, num, all) {

**console**.**log**("Word " + num + " in " + all.**toString**() + " is " + word);

});

**Prints:**  
Word 0 in cat,in,hat is cat\  
Word 1 in cat,in,hat is in\  
Word 2 in cat,in,hat is hat

***Did you know that...*** the toString() method converts and returns the value of an object or a variable into a string? For example, if you have a number or an array and you want to turn it into a readable string or manipulate it as text, you can use this method to achieve that.

On the next page, you'll do a quiz that uses the forEach() method to modify an array.

# Quiz: Another Type of Loop

## Directions:

Use the array's forEach() [method(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) to loop over the following array and add 100 to each of the values if the value is divisible by 3.

**const** test = [12, 929, 11, 3, 199, 1000, 7, 1, 24, 37, 4, 19, 300, 3775, 299, 36, 209, 148, 169, 299, 6, 109, 20, 58, 139, 59, 3, 1, 139];

Log your solution to the console.

#### Suggestion

You can test your code with a smaller array first like this one:

**const** miniTest = [12, 29, 11, 3];

// Use the push() method to add new crew members to the crew array

crew.push(doctor);

crew.push(sister);

crew.push(shepherd);

// Output the updated crew array

console.log(crew);

const test = [12, 929, 11, 3, 199, 1000, 7, 1, 24, 37, 4, 19, 300, 3775, 299, 36, 209, 148, 169, 299, 6, 109, 20, 58, 139, 59, 3, 1, 139];

// Create a new array to hold the modified values

const modifiedArray = [];

// Use forEach to loop over the array

test.forEach(value => {

    // Check if the value is divisible by 3

    if (value % 3 === 0) {

        // Add 100 to the value and push to the modified array

        modifiedArray.push(value + 100);

    } else {

        // If not divisible by 3, push the original value

        modifiedArray.push(value);

    }

});

// Log the modified array to the console

console.log(modifiedArray);

# Map

## How Should I Loop?

### for Loops

for loops are very versatile and give you complete control over the looping process because you can explicitly define where to start and stop in the array, whether you want to skip over values in the array, and whether you'd like to break out of the loop early using a break statement:

**const** myArray = [1, 2, 3, 4, 5];

**for** (**let** i = 0; i < myArray.length; i = i + 2) {

**console**.**log**(myArray[i]);

**if** (i === 2) {

**break**;

}

}

### forEach Loops

forEach gives you a concise way of looping over an array if you know you're going to be loopingover every element from start to finish. You have less versatility than with a regular for loop, but you can access each element directly, without using an index.

**const** myArray = [1, 2, 3, 4, 5];

myArray.**forEach**(**function** (elem) {

**console**.**log**(elem);

});

but using forEach() will not be useful if you want to permanently modify the original array. forEach() always returns undefined.

### map Loops

Creating a new array from an existing array is simple with the powerful map() method.

With the map() [method(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map), you can take an array, perform some operation on each element of the array, and return a new array.

**const** newArray = myArray.**map**(**function** (elem) {

elem = elem + 100;

**return** elem;

});

**console**.**log**(newArray); *//[101, 102, 103, 104, 105]*

#### Donut Example:

See how easy it is to create a new array of improved donuts?

**const** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

**const** improvedDonuts = donuts.**map**(**function**(donut) {

donut += " hole";

donut = donut.**toUpperCase**();

**return** donut;

});

**donuts array:** ["jelly donut", "chocolate donut", "glazed donut"]  
**improvedDonuts array:** ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]

The map() method accepts one argument, a function that will be used to manipulate each element in the array. In the above example, we used a function expression to pass that function into map(). This function is taking in one argument, donut which corresponds to each element in the donuts array. You no longer need to iterate over the indices anymore. map() does all that work for you.

# Quiz: I Got Bills

## Directions:

Use the map() [method(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) to take the array of bill amounts shown below, and create a second array of numbers called totals that shows the bill amounts with a 15% tip added.

**const** bills = [50.23, 19.12, 34.01, 100.11, 12.15, 9.90, 29.11, 12.99, 10.00, 99.22, 102.20, 100.10, 6.77, 2.22];

Print out the new totals array using console.log.

**TIP:** Check out the toFixed() method for numbers to help with [rounding(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) the values to a maximum of 2 decimal places. Note, that the method returns a string to maintain the "fixed" format of the number. So, if you want to convert the string back to a number, you can **cast** it or convert it back to a number:

// Log the modified array to the console

console.log(modifiedArray);

const bills = [50.23, 19.12, 34.01, 100.11, 12.15, 9.90, 29.11, 12.99, 10.00, 99.22, 102.20, 100.10, 6.77, 2.22];

// Use map to create a new array of totals with a 15% tip added

const totals = bills.map(function(bill) {

    // Calculate the total including a 15% tip

    const totalWithTip = bill + (bill \* 0.15);

    // Round to two decimal places and convert to number

    return Number(totalWithTip.toFixed(2));

});

// Print the new totals array

console.log(totals);

# Arrays in Arrays

## Think of an Array in an Array as a Grid

An array of arrays can be represented as a grid-like structure, where each element of the array corresponds to a row of cells.

* each row of the grid is an array
* each of those arrays is an element of the larger array

**const** grid = [

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

[ 6, 7, 8, 9, 10],

[11, 12, 13, 14, 15],

[16, 17, 18, 19, 20],

];

You can use a single loop to access each row of the grid.

**for** (**let** r = 0; r < grid.length; r++) {

**console**.**log**(grid[r]);

}

And you can use nested loops, or a loop inside of a loop, to loop over each element of each of those rows.

**for** (**let** r = 0; r < grid.length; r++) {

**for** (**let** c = 0; c < grid[r].length; c++) {

**console**.**log**(grid[r][c]);

}

}

The outer loop will cycle through each row and then in the row that's currently being iterated over, the inner loop will iterate over each cell.

***Best practice***: use intuitive variable names for your variables, like row and column or r and c . This will help you visualize what the nested array.

# 2D Donut Arrays

## A box of donuts is like a two-dimensional grid with rows and columns

You could use an array of arrays that has the name of each donut associated with its position in the box.

Here's an example:

**const** donutBox = [

["glazed", "chocolate glazed", "cinnamon"],

["powdered", "sprinkled", "glazed cruller"],

["chocolate cruller", "Boston creme", "creme de leche"]

];

If you wanted to loop over the donut box and display each donut (along with its position in the box!) you would start with writing a for loop to loop over each row of the box of donuts:

**const** donutBox = [

["glazed", "chocolate glazed", "cinnamon"],

["powdered", "sprinkled", "glazed cruller"],

["chocolate cruller", "Boston creme", "creme de leche"]

];

*// here, donutBox.length refers to the number of rows of donuts*

**for** (**let** row = 0; row < donutBox.length; row++) {

**console**.**log**(donutBox[row]);

}

**Prints:**

["glazed", "chocolate glazed", "cinnamon"]

["powdered", "sprinkled", "glazed cruller"]

["chocolate cruller", "Boston creme", "creme de leche"]

Since each row is an array of donuts, you next need to set up an inner-loop to loop over each cell in the arrays.

**for** (**let** row = 0; row < donutBox.length; row++) {

*// here, donutBox[row].length refers to the length of the donut array currently being looped over*

**for** (**let** column = 0; column < donutBox[row].length; column++) {

**console**.**log**(donutBox[row][column]);

}

}

**Prints:**

"glazed"

"chocolate glazed"

"cinnamon"

"powdered"

"sprinkled"

"glazed cruller"

"chocolate cruller"

"Boston creme"

"creme de leche"

In this loop, the outer loop cycles rows and the inner loops cycles through columns.

# Quiz: Nested Numbers

## Directions:

Use a nested for loop to take the numbers array below and replace all of the values that are divisible by 2 (even numbers) with the string "even" and all other numbers with the string "odd".

**var** numbers = [

[243, 12, 23, 12, 45, 45, 78, 66, 223, 3],

[34, 2, 1, 553, 23, 4, 66, 23, 4, 55],

[67, 56, 45, 553, 44, 55, 5, 428, 452, 3],

[12, 31, 55, 445, 79, 44, 674, 224, 4, 21],

[4, 2, 3, 52, 13, 51, 44, 1, 67, 5],

[5, 65, 4, 5, 5, 6, 5, 43, 23, 4424],

[74, 532, 6, 7, 35, 17, 89, 43, 43, 66],

[53, 6, 89, 10, 23, 52, 111, 44, 109, 80],

[67, 6, 53, 537, 2, 168, 16, 2, 1, 8],

[76, 7, 9, 6, 3, 73, 77, 100, 56, 100]

];

var numbers = [

  [243, 12, 23, 12, 45, 45, 78, 66, 223, 3],

  [34, 2, 1, 553, 23, 4, 66, 23, 4, 55],

  [67, 56, 45, 553, 44, 55, 5, 428, 452, 3],

  [12, 31, 55, 445, 79, 44, 674, 224, 4, 21],

  [4, 2, 3, 52, 13, 51, 44, 1, 67, 5],

  [5, 65, 4, 5, 5, 6, 5, 43, 23, 4424],

  [74, 532, 6, 7, 35, 17, 89, 43, 43, 66],

  [53, 6, 89, 10, 23, 52, 111, 44, 109, 80],

  [67, 6, 53, 537, 2, 168, 16, 2, 1, 8],

  [76, 7, 9, 6, 3, 73, 77, 100, 56, 100]

];

// Loop through each sub-array

for (let row = 0; row < numbers.length; row++) {

  // Loop through each element in the sub-array

  for (let col = 0; col < numbers[row].length; col++) {

      // Check if the current number is even or odd

      if (numbers[row][col] % 2 === 0) {

          numbers[row][col] = 'even'; // Replace even numbers with "even"

      } else {

          numbers[row][col] = 'odd'; // Replace odd numbers with "odd"

      }

  }

}

// Print the final array

console.log(numbers);

# Lesson Summary

## What We Learned in This Lesson

You now know all about array!

* How to create arrays
* How to use arrays
* How arrays are structured

## On to Objects!

In the next lesson of this course, you'll be expl

# Intro to Objects

## Objects are Powerful!

Objects allow us to wrap up pieces of related data and functionality into one single container.

Arrays are a special type of object in JavaScript with methods including length() , .reverse() , sort() , push(), and pop() -- but objects are much more than arrays. JavaScript includes a lot of other built-in objects.

In this lesson, we'll learn all about how we can create our own objects that contain both data and functions that we can use to manipulate or operate on that data. By the end of this lesson you will be able to:

* Create a JavaScript object
* Use dot notation and bracket notation to access object elements
* Use typeof to access the type of a variable
* Add and access methods in JavaScript objects
* Use JavaScript naming conventions for property names

Let's get started!

# Objects in Code

## Follow Along with the Demo!

### Using typeof

typeof is handy operator that returns the name of the data type that follows it. Try running this code in your console:

**typeof** "hello" *// returns "string"*

**typeof** true *// returns "boolean"*

**typeof** [1, 2, 3] *// returns "object" (Arrays are a type of object)*

**typeof** **function** **hello**() { } *// returns "function"*

### Creating an Object

Paste this JavaScript code in the console to create the object:

**const** umbrella = {

color: "pink",

isOpen: false,

**open**: **function**() {

**if**(umbrella.isOpen === true) {

**return** "The umbrella is already opened!";

} **else** {

umbrella.isOpen = true;

**return** "Julia opens the umbrella!";

}

}

}

### Using the Object

Then run this code, one line at a time:

umbrella.isOpen; *// returns false*

umbrella.**open**(); *// returns 'Julia opens the umbrella!'*

umbrella.isOpen; *// returns true*

We'll go into more detail about to create and use objects soon!

### Reminder -- var vs let and const

As discussed in earlier lessons, it is now considered best practice to use let or const instead of var.

Objects are [***JavaScript Object types***(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects) that are assigned by reference, not by value.

You'll see both var and const used to declare objects in this lesson, but in almost all cases you'll want to use const to declare objects in your own code.

# Quiz: Umbrella

## Directions:

Using the umbrella example from the previous video, see if you can follow the example open() method and create the close() method. It's alright if you have trouble at first! We'll go into more detail later in this lesson.

**const** umbrella = {

color: "pink",

isOpen: false,

**open**: **function**() {

**if** (umbrella.isOpen === true) {

**return** "The umbrella is already opened!";

} **else** {

umbrella.isOpen = true;

**return** "Julia opens the umbrella!";

}

}

};

**TIP:** Remember to put all of your object's properties and methods inside curly braces: const myObject = { greeting: "hello", name: "Julia" };. Also, remember that an object is just another data type. Just like how you would put a semicolon after a string variable declaration const myString = "hello";, don't forget to put a semi-colon at the end of your object's declaration.

// Define the umbrella object

const umbrella = {

  color: "pink",

  isOpen: false,

  open: function() {

    if (this.isOpen) {

      return "The umbrella is already opened!";

    } else {

      this.isOpen = true;

      return "Julia opens the umbrella!";

    }

  },

  close: function() {

    if (!this.isOpen) {

      return "The umbrella is already closed!";

    } else {

      this.isOpen = false;

      return "Julia closes the umbrella!";

    }

  }

};

// Test the methods

console.log(umbrella.open());  // Outputs: Julia opens the umbrella!

console.log(umbrella.open());  // Outputs: The umbrella is already opened!

console.log(umbrella.close()); // Outputs: Julia closes the umbrella!

console.log(umbrella.close()); // Outputs: The umbrella is already closed!

# Objects

## What Is An Object?

An object is a data structure in JavaScript that let's you store data about a particular thing and helps you keep track of that data using a "key".

### Person Example

Defining data about a person in isolated variables is clunky:

const sisterName = "Sarah";

const sisterAge = 23;

const sisterParents = ["Alice", "Andy"];

const sisterSiblings = ["Julia"];

const sisterFavoriteColor = "purple";

const sisterPets = true;

An object allows you group this information into a meaningful structure:

**const** sister = {

name: "Sarah",

age: 23,

parents: ["Alice", "Andy"],

siblings: ["Julia"],

favoriteColor: "purple",

pets: true

};

# Object Literals

## Object-literal notation

**const** sister = {

name: "Sarah",

age: 23,

parents: [ "Alice", "Andy" ],

siblings: ["julia"],

favoriteColor: "purple",

pets: true

};

The syntax you see above is called **object-literal notation**. There are some important things you need to remember when you're structuring an object literal:

* The "key" (representing a **property** or **method** name) and its "value" are separated from each other by a **colon**
* The key: value pairs are separated from each other by **commas**
* The entire object is wrapped inside curly braces { }.

And, kind of like how you can look up a word in the dictionary to find its definition, the key in a key:value pair allows you to look up a piece of information about an object. Here's are a couple examples of how you can retrieve information about my sister's parents using the object you created.

*// two equivalent ways to use the key to return its value*

sister["parents"] *// returns ["Alice", "Andy" ]*

sister.parents *// also returns ["Alice", "Andy"]*

Using sister["parents"] is called **bracket notation** (because of the brackets!) and using sister.parents is called **dot notation** (because of the dot!).

## What about methods?

The sister object above contains a bunch of properties about my sister, but doesn't really say what my sister does. For instance, let's say my sister likes to paint. You might have a paintPicture() method that returns "Sarah paints a picture!" whenever you call it. The syntax for this is pretty much exactly the same as how you defined the properties of the object. The only difference is, the value in the key:value pair will be a function.

**const** sister = {

name: "Sarah",

age: 23,

parents: ["Alice", "Andy"],

siblings: ["Julia"],

favoriteColor: "purple",

pets: true,

**paintPicture**: **function**() { **return** "Sarah paints!"; }

};

sister.**paintPicture**();

**Returns**: "Sarah paints!"

and you can access the name of my sister by accessing the name property:

sister.name

**Returns**: "Sarah"

# Naming Conventions

## Best Practices for Naming Object Properties

### Using Quotes

Using quotes around property names is valid but not required -- but be careful! Sometimes quotes can mask naming problems with dot notation.

### Problem 1: Using a number as the first character in a property name

**const** person = {

"name": "John",

"age": 55,

"1stChild": "James", *// USes a number as the first character in a property name*

"2ndChild": "Jarrod",

"3rdChild": "Alexis"

};

person["1stChild"]; *// returns "James"*

person.1stChild; *// returns Uncaught SyntaxError: Invalid or unexpected token*

It would be even better to store the children in an array:

**const** person = {

name: "John",

age: 55,

children: ["James", "Jarrod","Alexis"]

};

### Problem 2: Using spaces or hyphens in a property name

**const** garage = {

"fire truck": {

"color": "red",

"wheels": 4,

"operational": true

},

"race-car": {

"color": "blue",

"wheels": 4,

"operational": false

},

};

garage.fire truck; *// returns Uncaught SyntaxError: Unexpected identifier*

garage.race-car; *// returns Uncaught ReferenceError: car is not defined*

You can avoid these issues by using camelCase when you want a multi-word variable name:

**const** garage = {

"fireTruck": {

"color": "red",

"wheels": 4,

"operational": true

},

"raceCar": {

"color": "blue",

"wheels": 4,

"operational": false

},

};

## Naming Best Practices

* Don't use a number as the first character in a property name
* Don't use quotes around key names unless absolutely necessary
* Use camelCase when you need a multi-word variable name.

These naming conventions also apply to regular variable names too!

# Summary of Objects

Objects are one of the most important data structures in JavaScript. Get ready to see them everywhere!

They have properties (information about the object) and methods (functions or capabilities the object has). Objects are an incredibly powerful data type and you will see them all over the place when working with JavaScript, or any other object-oriented programming language.

## Object literals, methods, and properties

You can define objects using **object-literal notation**:

**const** myObj = {

color: "orange",

shape: "sphere",

type: "food",

**eat**: **function**() { **return** "yummy" }

};

myObj.**eat**(); *// method*

myObj.color; *// property*

## Naming conventions

Feel free to use upper and lowercase numbers and letters, but don't start your property name with a number. You don't need to wrap the string in quotes! If it's a multi-word property, use camel case. Don't use hyphens in your property names

**const** richard = {

"1stSon": true,

"loves-snow": true

};

richard.1stSon *// error*

richard.loves-snow *// error*

# Quiz: Menu Items

## Directions:

Create a breakfast object to represent the following menu item:

The Lumberjack - $9.95

eggs, sausage, toast, hashbrowns, pancakes

The object should contain properties for the name, price, and ingredients.

const breakfast = {

  name: "The Lumberjack",

  price: "$9.95",

  ingredients: ["eggs", "sausage", "toast", "hashbrowns", "pancakes"]

};

console.log(breakfast);

# Quiz: Bank Accounts 1

## Directions:

Using the given object:

**var** savingsAccount = {

balance: 1000,

interestRatePercent: 1,

**deposit**: **function** **addMoney**(amount) {

**if** (amount > 0) {

savingsAccount.balance += amount;

}

},

**withdraw**: **function** **removeMoney**(amount) {

**var** verifyBalance = savingsAccount.balance - amount;

**if** (amount > 0 && verifyBalance >= 0) {

savingsAccount.balance -= amount;

}

}

};

add a printAccountSummary() method that returns the following account message:

Welcome!

Your balance is currently $1000 and your interest rate is 1%.

var savingsAccount = {

  balance: 1000,

  interestRatePercent: 1,

  deposit: function addMoney(amount) {

    if (amount > 0) {

      savingsAccount.balance += amount;

    }

  },

  withdraw: function removeMoney(amount) {

    var verifyBalance = savingsAccount.balance - amount;

    if (amount > 0 && verifyBalance >= 0) {

      savingsAccount.balance -= amount;

    }

  },

  printAccountSummary: function() {

    return `Welcome! Your balance is currently $${this.balance} and your interest rate is ${this.interestRatePercent}%.`;

  }

};

// Example usage

console.log(savingsAccount.printAccountSummary());

# Quiz: Facebook Friends

## Directions:

Create an object called facebookProfile. The object should have 3 properties:

1. your name
2. the number of friends you have, and
3. an array of messages you've posted (as strings) -- include at least two messages.

The object should also have 4 methods:

1. postMessage(message) - adds a new message string to the array
2. deleteMessage(index) - removes the message corresponding to the index provided
3. addFriend() - increases the friend count by 1
4. removeFriend() - decreases the friend count by 1

***HINT!*** Here are some array methods you might want to use:

* addition at the end is done using the push() method - [MDN: Array.prototype.push()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push)
* addition or removal at a specific index is done using the splice() method - [Array.prototype.splice()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice)
* deletion from the end is done using the pop() method - [MDN: Array.prototype.pop()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop)

# Creating an object in JavaScript that models a Facebook profile involves using object-literal notation to define properties and methods. Below is an example of how to create a facebookProfile object with the required properties and methods, followed by a breakdown of the concepts involved.

// Define the facebookProfile object

const facebookProfile = {

  name: "Udacian", // Property: user's name

  friendsCount: 25, // Property: number of friends

  messages: [ // Property: array of messages

      "Message 1",

      "Message 2"

  ],

  // Method to post a new message

  postMessage: function(message) {

      this.messages.push(message); // Adds a new message to the messages array

      console.log("Messages:", this.messages); // Log current messages

  },

  // Method to delete a message by index

  deleteMessage: function(index) {

      if (index >= 0 && index < this.messages.length) { // Check for valid index

          this.messages.splice(index, 1); // Removes the message at the specified index

          console.log("Messages:", this.messages); // Log current messages

      } else {

          console.log("Invalid index."); // Log a message if index is invalid

      }

  },

  // Method to add a friend

  addFriend: function() {

      this.friendsCount += 1; // Increases the friends count by 1

      console.log("Friends:", this.friendsCount); // Log current friends count

  },

  // Method to remove a friend

  removeFriend: function() {

      if (this.friendsCount > 0) { // Check to ensure friends count doesn't go negative

          this.friendsCount -= 1; // Decreases the friends count by 1

          console.log("Friends:", this.friendsCount); // Log current friends count

      } else {

          console.log("No friends to remove."); // Log a message if no friends are left

      }

  }

};

// Example Usage:

// Run the following commands to see how the methods work

facebookProfile.postMessage("New message!"); // Adding a message

facebookProfile.deleteMessage(1); // Deleting the second message

facebookProfile.addFriend(); // Adding a friend

facebookProfile.removeFriend(); // Removing a friend

# Quiz: Donuts Revisited

Here is an array of donut objects.

**const** donuts = [

{ type: "Jelly", cost: 1.22 },

{ type: "Chocolate", cost: 2.45 },

{ type: "Cider", cost: 1.59 },

{ type: "Boston Cream", cost: 5.99 }

];

## Directions:

Use the forEach() method to loop over the array and print out the following donut summaries using console.log.

Jelly donuts cost $1.22 each

Chocolate donuts cost $2.45 each

Cider donuts cost $1.59 each

Boston Cream donuts cost $5.99 each

If you need a refresher on the forEach method, check out the documentation: [MDN - Array.prototype.forEach()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)

### Running Your Code

Enter the following in the terminal:

**node** donuts.js

When your code runs, you should get this output:

Jelly donuts cost $1.22 each

Chocolate donuts cost $2.45 each

Cider donuts cost $1.59 each

Boston Cream donuts cost $5.99 each

// Example Usage:

// Run the following commands to see how the methods work

facebookProfile.postMessage("New message!"); // Adding a message

facebookProfile.deleteMessage(1); // Deleting the second message

facebookProfile.addFriend(); // Adding a friend

facebookProfile.removeFriend(); // Removing a friend

// Array of donut objects

const donuts = [

  { type: "Jelly", cost: 1.22 },

  { type: "Chocolate", cost: 2.45 },

  { type: "Cider", cost: 1.59 },

  { type: "Boston Cream", cost: 5.99 }

];

// Using forEach to loop over the array and print the summaries

donuts.forEach(function(donut) {

  console.log(`${donut.type} donuts cost $${donut.cost} each`);

});

# Lesson Summary

## Congratulations!

You have completed this introductory course on JavaScript! From defining your first variables, to writing functions,to creating your very own objects,you've come a long way.

### What We Learned in This Lesson

You now know a lot about JavaScript objects! You can:

* Create a JavaScript object
* Use dot notation and bracket notation to access object elements
* Use typeof to access the type of a variable
* Add and access methods in JavaScript objects
* Use JavaScript naming conventions for property name

## What We Learned in This Course

In just a short amount of time, you've built up a basic understanding of the JavaScript programming language, and how it works.

You now know how to:

* Run JavaScript code in the console of your web brow
* Create variables and use basic JavaScript data types to represent real-world data
* Use conditionals to add logic to your JavaScript programs
* Create loops to reduce code duplication and add automation
* Write functions to streamline and organize your code
* Use arrays to store and manipulate lists of data
* Use objects for even more complex data organization

Great work!!!

**The document object model**

# Lesson Intro

## It's All About the DOM!

In this lesson, we'll learn all about the Document Object Model a.k.a. the DOM.

We'll start with a discussion what the DOM is and how it gets created.

Next, we'll learn several different JavaScript methods for programmatically accessing DOM elements:

* .getElementById()
* .getElementsByClassName()
* .getElementsByTagName()
* .querySelector()
* .querySelectorAll()

We'll also learn about the web interfaces that facilitate this access, including:

* Element
* Node

We have a lot to cover in this lesson. Let's get started!

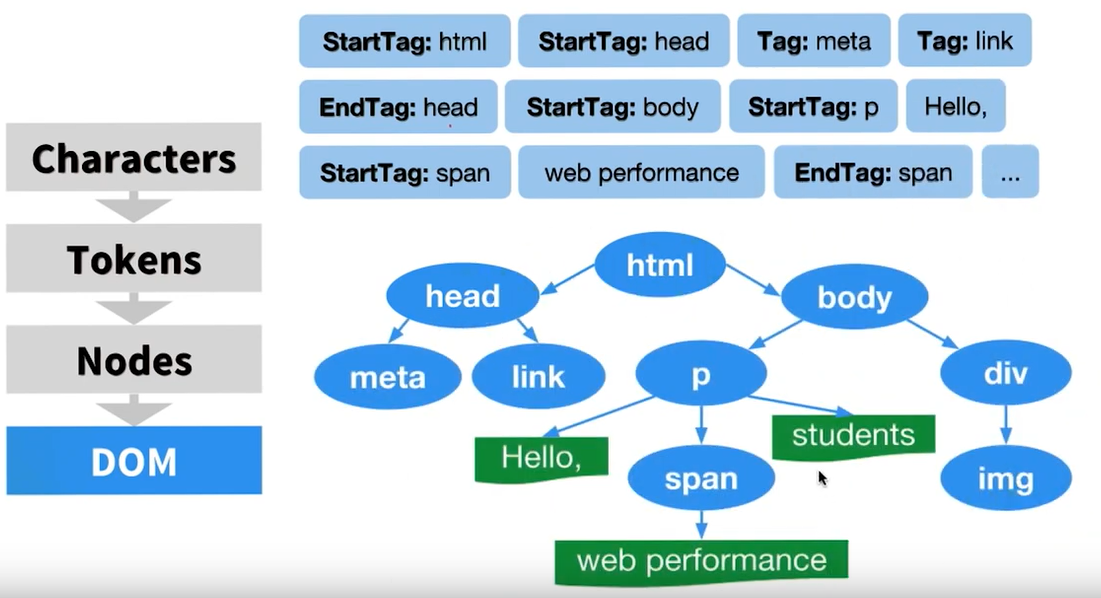
## The DOM

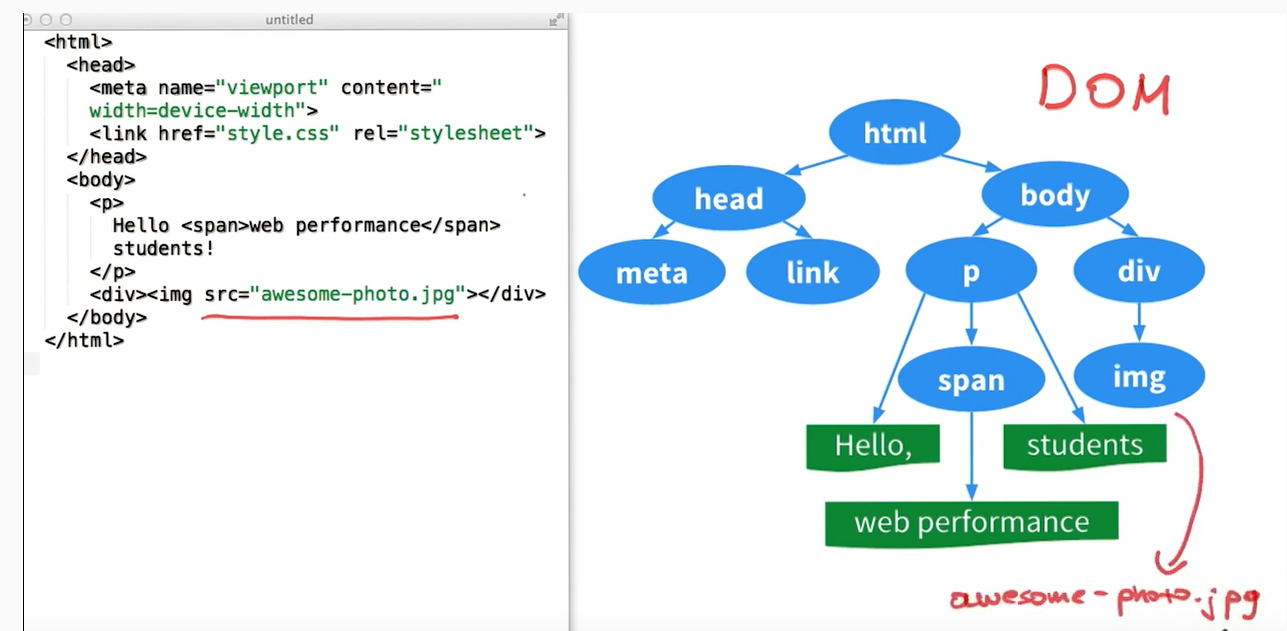
In this section, we'll look at the **Document Object Model** - otherwise known as the DOM.

The words "the DOM" are used all over developer documentation sites and tutorials on writing interactive JavaScript code. But what is it? Perhaps you've even used the DOM and still aren't quite sure what it is. Is it the browser? Is it a special part of JavaScript? ¯\\_(ツ)\_/¯

### Reflect

Take a couple of minutes think about what you think "the DOM" is/refers to, and write out a description of the DOM.





### Translating HTML to the DOM

When you request a website, no matter what backend language is powering that website, it will respond with HTML. This response is based on the HTML specification which contains a specific set of rules for how browsers should process the data they recieve.

The process works in this order:

1. HTML is received
2. HTML tags are converted to tokens
3. Tokens are converted to Nodes
4. Nodes are converted to the DOM

#### The Details

The browser receives a stream of HTML. The bytes are run through a complicated (but fully documented) parsing process that determines the different characters (e.g. the start tag character <, an attribute like href, a closing angle bracket like >). After parsing has occurred, a process called **tokenization** begins. Tokenization takes one character at a time and builds up **tokens**. The tokens are:

* DOCTYPE
* start tag
* end tag
* comment
* character
* end-of-file

Let's take a break for a second. At this state, the browser has received the bytes that've been sent by a server. The browser has converted the bytes to tags and has read through the tags to create a list of tokens.

This list of tokens then goes through the tree construction stage. The output of this stage is a tree-like structure - this is the DOM!

Two important quotes from Illya in the video:

a tree structure that captures the content and properties of the HTML and all the relationships between the nodes

the DOM is the full, parsed representation of the HTML

So the DOM is a model (representation) of the relationships and attributes of the HTML document that was received. Remember that DOM stands for "Document Object Model". Something that I've found to be true as I've been learning is that to break something down, just read the thing backwards:

Document Object Model

...would become…

Object Model of the Document!

Remember that a JavaScript object is a tree-like structure that has properties and values. So the DOM can be accessed using a special object provided by the browser: document

### Exploring the DOM

Try this:

1. Open the console on this page
2. Type out the word document
   * careful not to declare it (const document)
   * careful not to wrap it in quotes ("document")
3. Press enter

The document object is provided by the browser and is a representation of the HTML document. This object is not provided by the JavaScript language. ECMAScript is the language specification that JavaScript is based on, and it only references the document object model in one place, in its "Global Object" section:

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself; for example, in the HTML document object model the window property of the global object is the global object itself. ([source(opens in a new tab)](https://www.ecma-international.org/ecma-262/#sec-global-object))

Basically, this says that the document object is not part of JavaScript, but is expected to already exist and be freely accessible to JavaScript code.

The DOM is standardized by the W3C. There are a number of specifications that make up the DOM, here are few:

* Core Specification
* Events Specification
* Style Specification
* Validation Specification
* Load and Save Specification

To see the full list of DOM specs, check out the standard at: <https://www.w3.org/standards/techs/dom#w3c_all>

## The DOM Recap

The DOM stands for "Document Object Model" and is a tree-like structure that is a representation of the HTML document, the relationship between elements, and contains the content and properties of the elements.

The DOM is not:

* part of the JavaScript language

The DOM is:

* constructed from the browser
* is globally accessible by JavaScript code using the document object

The DOM is used all of the time and is what we'll be using throughout this course!

### Further Research

* [DOM Introduction(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)
* [Section 8.2 Parsing HTML documents(opens in a new tab)](https://www.w3.org/TR/html5/syntax.html#parsing) from the W3C's HTML Documentation
* [DOM Specification(opens in a new tab)](https://www.w3.org/standards/techs/dom#w3c_all) on W3C
* [HTML Document Object Model mentioned in the ECMAScript Specification(opens in a new tab)](https://www.ecma-international.org/ecma-262/#sec-global-object) - the language specification used by JavaScript

# Solution: Exploring the DOM

## The #document Object Looks Like HTML!

Exploring the DOM

The document object is provided by the browser and is a ***representation*** of the HTML document.

This object is not provided by the JavaScript language. ECMAScript is the language specification that JavaScript is based on, and it only references the document object model in one place, in its "Global Object" section:

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself; for example, in the HTML document object model the window property of the global object is the global object itself. ([source(opens in a new tab)](https://www.ecma-international.org/ecma-262/#sec-global-object))

Basically, this says that the document object is not part of JavaScript, but is expected to already exist and be freely accessible to JavaScript code.

The DOM is standardized by the W3C. There are a number of specifications that make up the DOM, here are few:

The DOM is standardized by the W3C. There are a number of specifications that make up the DOM, here are few:

* Core Specification
* Events Specification
* Style Specification
* Validation Specification
* Load and Save Specification

To see the full list of DOM specs, check out the standard at: <https://www.w3.org/standards/techs/dom#w3c_all>

# Recap: The DOM

## Key Points to Remember

The DOM stands for "Document Object Model" and is a tree-like structure that is a representation of the HTML document, the relationship between elements, and contains the content and properties of the elements.

The DOM is not:

* part of the JavaScript language

The DOM is:

* constructed from the browser
* is globally accessible by JavaScript code using the document object

The DOM is used all of the time and is what we'll be using throughout this course!

### Further Research

* [DOM Introduction(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)
* [W3.org: What is the Document Object Model?(opens in a new tab)](https://www.w3.org/TR/WD-DOM/introduction.html)
* [Section 8.2 Parsing HTML documents(opens in a new tab)](https://www.w3.org/TR/html5/syntax.html#parsing) from the W3C's HTML Documentation
* [DOM Specification(opens in a new tab)](https://www.w3.org/standards/techs/dom#w3c_all) on W3C
* [HTML Document Object Model mentioned in the ECMAScript Specification(opens in a new tab)](https://www.ecma-international.org/ecma-262/#sec-global-object) - the language specification used by JavaScript

# CSS Review

## A CSS Mindset

You're probably used to seeing (and writing!) a lot of this kinda stuff:

.header {

max-width: 100%;

margin-left: auto;

margin-right: auto

}

.header.header--clone .header\_\_navbar {

height: 4.5rem

}

.header.header--stick {

top: 0;

background-color: #ffffff;

z-index: 8000;

box-shadow: 0px 1px 2px **rgba**(0,0,0,0.1)

}

.header\_\_navbar {

max-width: 73.75rem;

margin-left: auto;

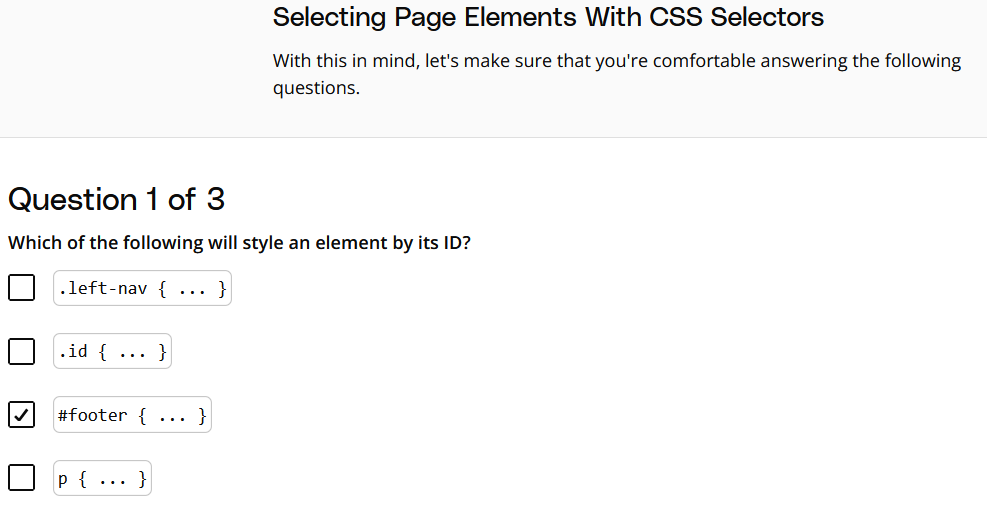
margin-right: auto;

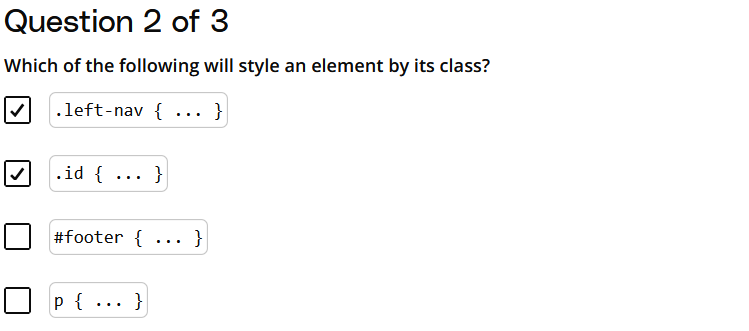
height: 4.5rem;

padding: 0 1.5em

}

...you know, CSS! The skills that you've learned writing CSS will help you in this section!





In CSS:

* IDs are selected with a **hash symbol (#)**.
* Classes are selected with a **dot symbol (.)**.
* Tag names are written directly without any prefix.

So this was a quick review on how to select elements by ID, class, and tag. Believe it or not, being able to select HTML elements this way is actually going to be a vital skill in this section where we learn how to access page elements using JavaScript and the DOM!

If you struggled a little bit with this section, perhaps you should do a quick CSS review before you move on to the rest of this lesson.

Alrighty - let's get right to it and let the fun begin

# Select Page Element By ID

## Select An Element By ID

Let's take a look at how we can use JavaScript and the DOM to gain access to specific elements using their ID attribute.

Remember the document object from the previous section? Well, we're going to start using it! Remember the document object is an object, just like a JavaScript object. This means it has key/value pairs. Some of the values are just pieces of data, while others are functions (also known as **methods**!) that provide some type of functionality. The first DOM method that we'll be looking at is the .getElementById() method:

document.**getElementById**();

If we ran the code above in the console, we wouldn't get anything, because we did not tell it the ID of any element to get! We need to pass a string to .getElementById() of the ID of the element that we want it to find and subsequently return to us:

document.**getElementById**('footer');

One thing to notice right off the bat, is that we're passing 'footer', not '#footer'. It already knows that it's searching for an ID (its name is "getElementById", for a reason!).

If you'd like to read more about this method, check out its documentation page on MDN: [Document.getElementById()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById)

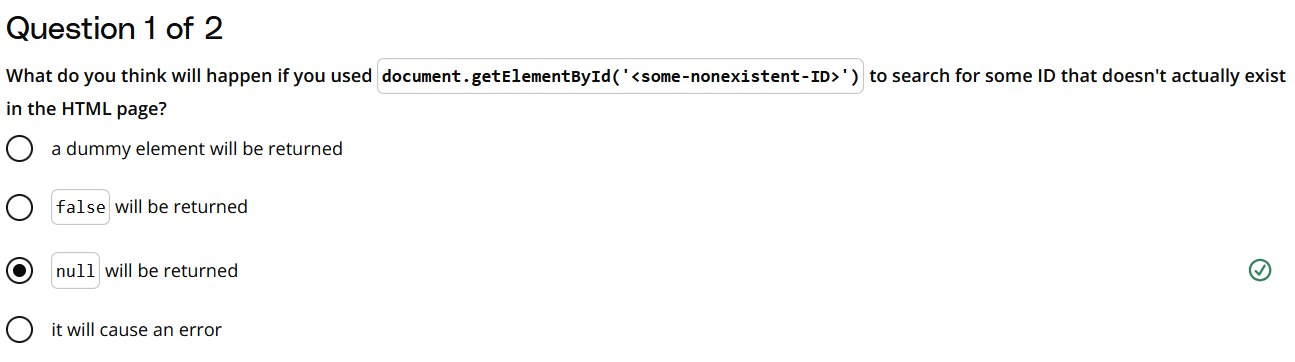
Let's use this MDN documentation page to try out using this method.

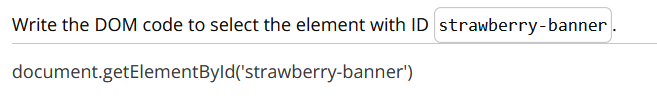
👉 ***NOTE:*** MDN Web Docs has updated their styling. The pages now look different, but the page structure is still the same

### Demo Recap

* we opened the DevTools for the page we were looking at
* we switched to the Console pane
* we ran document.getElementById('content'); on the console

Running this code caused the document object to search through its entire tree-like structure for the element that has an ID of "content".





### Storing a Selected Element for Use Later

### Demo Recap

We used document.getElementByID to store an element to access later, just like we can with any JavaScript function:

**const** pageContent = document.**getElementById**('content');

We'll learn how to use elements we've stored in variables later lessons!

### Demo Recap

We used document.getElementByID to store an element to access later, just like we can with any JavaScript function:

**const** pageContent = document.**getElementById**('content');

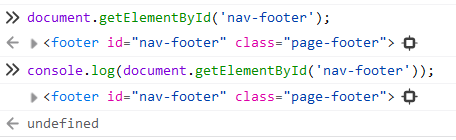
We'll learn how to use elements we've stored in variables later lessons!

# Exercise: Document.getElementById()

## Selecting a Page Element Using its ID

Now it's your turn! Use the document.getElementById() method to find the element that has the syntax id, save it in a variable named syntaxElement, and look at what was returned.

Selecting by ID and store…..



# Solution: Document.getElementById()

Find the element with the ID *syntax* using document.getElementById();

To recap what we just did. We:

* Opened the DevTools for the page we were looking at
* Switched to the ***Console*** pane
* Declared a variable syntaxElement in the console
* Assigned the value of document.getElementById('syntax') to syntaxElement
* logged out the value of syntaxElement to see what we returned

**const** syntaxElement = document.**getElementById**('syntax');

**console**.**log**(syntaxElement);

Running this code caused the document object to search through its entire tree-like structure for the element that has an ID of "syntax" and then save the returned element for us!

# Recap: Document.getElementById()

## Selecting By ID Recap

In this section, we learned how to select a DOM element by its ID:

*// select the element with the ID "idName"*

document.**getElementById**('idName');

We also learned how to save that selected element so we can use it later:

*// save the element with the ID "idName"in the variable "idElement"*

**const** idElement = document.**getElementById**('idName');

There are a couple of important things to keep in mind about this method:

* it is called on the document object
* it returns a single item

### Further Research

* [Document.getElementById()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById) on MDN

# Select Page Elements By Class Or Tag

## Selecting Multiple Elements At Once

As I'm sure you remember from learning both HTML structure and CSS styling, an ID should be unique - meaning two or more elements should never have the same ID. Since IDs are unique, and since there will be only one element in the HTML with that ID, document.getElementById() will only ever return at most one element. So how would we select multiple DOM elements?

The next two DOM methods that we'll be looking at that both return multiple elements are:

* .getElementsByClassName()
* .getElementsByTagName()

## Accessing Elements By Their Class

The first method we'll look at is .getElementsByClassName():

document.**getElementsByClassName**();

Similarly to .getElementById(), if we ran the code above in the console, we wouldn't get anything, because we did not tell it the class to search for! Also just like .getElementById(), .getElementsByClassName() is expecting that we call it with a string of the class we want it to search for/return:

document.**getElementsByClassName**('brand-color');

If you'd like to read more about this method, check out its documentation page on MDN: [Document.getElementsByClassName()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByClassName)

Let's use this MDN documentation page to try out using this method.

### Demo Recap

In this demo, we:

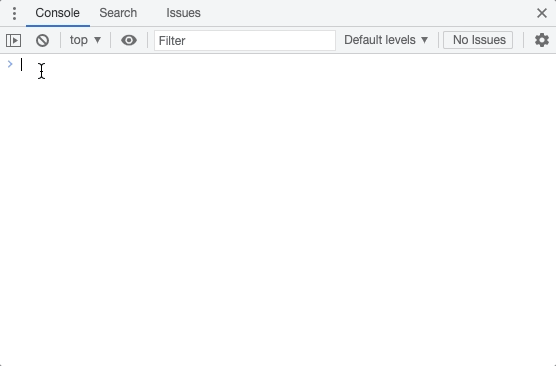
1. Navigated to the MDN [Documents.getElementsByClassName()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByClassName) page.
2. Right clicked on an element and selected ***Inspect*** which opened up the Dev Tools console.
3. Used document.getElementsByClassName in the console to select all of the elements with the class highlight-spanned:

document.**getElementsByClassName**('highlight-spanned');

1. Reviewed the items returned.

***Note***: The MDN UI has been updated and no longer uses the highlight-spanned class. If you want to try this on your own, look for elements with the section-content class:

document.**getElementsByClassName**('section-content');



Accessing elements with the *section-content* class using *document.getElementsByClassName()*

## Accessing Elements By Their Tag

After looking at both .getElementById() and .getElementsByClassName(), the new .getElementsByTagName() method should seem quite easy on the eyes:

document.**getElementsByTagName**('p');

Let's use this MDN documentation page to try out using this method: [(opens in a new tab)https://developer.mozilla.org/en-US/docs/Web/API/Document\_Object\_Model/Introduction(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

### Demo Recap

In this demo, we:

1. Navigated to the MDN [Introduction to the DOM()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction) page.
2. Right clicked on a code block on the page and selected ***Inspect*** which opened up the Dev Tools console.
3. Used document.getElementsByTagName in the console to select all of the elements with the pre tag:

document.**getElementsByTagName**('pre');

1. Reviewed the items returned.
2. Right-clicked on one of the elements and chose ***Reveal in Element panel***.
3. Right-clicked on the element in the ***Elements*** panel and selected ***Scroll into view*** to see the element on the page.

## What's Returned?

You might have noticed that .getElementsByClassName() and .getElementsByTagName() returned something that looks like an array -- but it's not an array! Let's look at it again:

### Demo Recap

In this demo, we:

1. Navigated to the MDN [Document.getElementsByClassName() (opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByClassName) page.
2. Right clicked on an code block on the page and selected ***Inspect*** which opened up the Dev Tools console.
3. Used document.getElementsByClassName in the console to select all of the elements with the line-numbers class:

document.**getElementsByClassName**('line-numbers');

1. Reviewed the items returned and learned that .getElementsByClassName() returns an array-like data structure of elements called an **HTMLCollection**.

.getElementsByTagName also returns an HTMLCollection.

You may be asking: what exactly is an element? In the next section, we'll take the plunge and look at Elements and Nodes.

***Note***: The MDN UI has been updated and no longer uses the line-numbers class.

If you want to try this on your own, right click on one of the code blocks and find a current class name and use that name instead!

### ⚠️ Beware of the S! ⚠️

There's something different about .getElementById() compared with both .getElementsByClassName() and .getElementsByTagName() that can be easy to miss; both .getElementsByClassName() and .getElementsByTagName() have an extra "s" in their name.

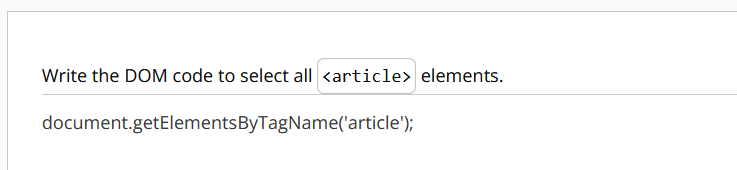
The method's name is .getElementsByClassName(), not .getElementByClassName(). Notice the word right in the middle, it's "Elements" not "Element". If you think about it, this actually makes a lot of sense! Since both .getElementsByClassName() and .getElementsByTagName() could return multiple items, their method names tell us that directly. Now compare this with .getElementById() that will only ever return at most one element. Its name has the singular "Element" in it.

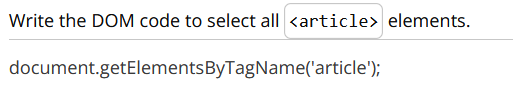
I just wanted to point this out because I've been bitten by that missing "s" many-a-time when running code like:

document.**getElementByClassName**('highlight-spanned');

This code above will not work, because there is no DOM method .getElementByClassName() (with singular "Element").

# Quizzes: Select Page Elements By Class Or Tag





# Recap: Select Page Elements By Class Or Tag

## Selecting Multiple Elements At Once Recap

In this section, we learned two ways to select multiple DOM elements:

* .getElementsByClassName()
* .getElementsByTagName()

There are a few important things to keep in mind about these two methods:

* both methods use the document object
* both return multiple items
* the list that's returned is an HTMLCollection, not an array

*// select all elements that have the class "accent-color"*

document.**getElementsByClassName**('accent-color');

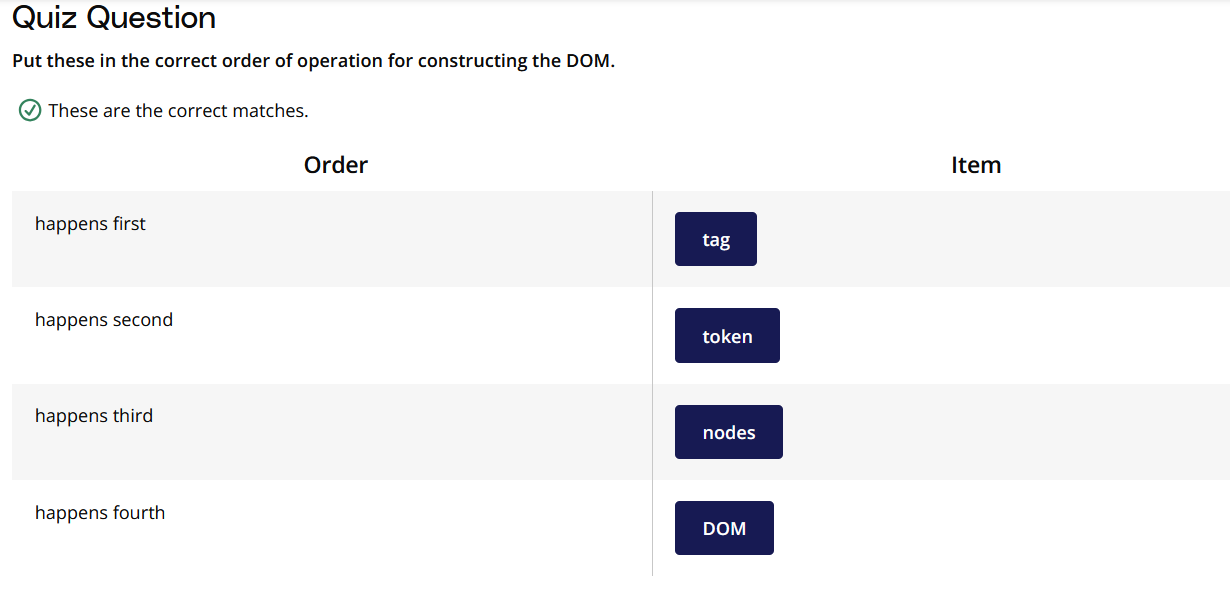
*// select all "span" elements*

document.**getElementsByTagName**('span');

### Further Research

* [Document.getElementsByClassName()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByClassName) on MDN
* [Document.getElementsByTagName()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Element/getElementsByTagName) on MDN

# Nodes, Elements, and Interfaces...Oh My!



So to reiterate the process, it's:

1. characters
2. tags
3. tokens
4. nodes
5. DOM

## But what is a "node"???

Great question! Let's figure this out.

### What's the difference between "Node" and "node"?

If you have experience with object-oriented programming, think about it this way:

* ***Node*** is like a ***class***.
* ***node*** is like an ***object***.

If you aren't familiar with object-oriented programming, think about it this way:

* ***Node*** is like the ***blueprint*** for a building.
* ***node*** is like an an ***actual building*** that is built using the blueprint.

Just like the blueprint for a building information about the features and capabilities of the building, a Node includes properties and methods of the actual node items.

Another word for blueprint is ***interface***. An interface gives us the properties and methods that are applied to the individual items.

To summarize:

* interface = blueprint
* properties = data
* methods = functionality

Don't worry if this is still a bit hazy. It will make more sense when you see it in action.

### ⚠️ Don't confuse Interface with User Interface ⚠️

The word "interface" might be an unclear word right now, and that's ok. I do want to make sure that you're not connecting this "interface" with a user interface (UI) or a graphical user interface (GUI).

Our use of "interface" is not related to either a UI or a GUI. Our use of "interface" is a technical, computer science word for a list of properties and methods that are inherited.

# The Node Interface

### Demo Recap

In this demo:

1. First, we navigated to the MDN [Node(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Node) page and scrolled the page to see the properties and methods.
2. In the browser navigation menu, we clicked on ***View***, then ***Developer*** and ***JavaScript Console***.
3. In the console, we stored the value of the body element in a variable named bodyEl:

**const** bodyEl = document.body;

1. We called the baseURI method on bodyEl, which returned the URI of the webpage:

bodyEl.baseURI

*// returns 'https://developer.mozilla.org/en-US/docs/Web/API/Node'*

1. We called the nodeType method on bodyEl, which returned the value 1, which indicates that the type of the body node is ELEMENT\_NODE:

bodyEl.nodeType

*// returns 1, which is the ELEMENT\_NODE type*

So the Node Interface is a blueprint for all of the properties (data) and methods (functionality) that every real node has after it's been created. Now, the Node Interface has a lot of properties and methods, but it's not very specific...I mean, what is a node???

Just like "blueprint for a Building" is not as specific as "blueprint for a house" or "blueprint for a skyscraper". These are more-specific blueprints. And these more-specific blueprints would probably have their own properties and methods that are specific to just houses or just skyscrapers.

Next, we'll look at the "Element Interface".

# The Element Interface

## Element Interface

Just like the Node Interface, the Element Interface is a blueprint for creating elements: [Element Interface on MDN(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Element)

One really important thing about the Element Interface is that it is a descendent of the Node Interface.



Element points to its parent, Node.

Since Element is pointing at Node, this indicates that the Element Interface inherits all of the Node Interface's properties and methods. This means that any element (lowercase "e"!) that was created from the Element Interface is also a descendent from the Node Interface...which means the element (lowercase "e"!) is also a node (lowercase "n"!).

Let's do a little digging around on an element!

### Demo Recap

In this demo:

1. First, we navigated to the MDN [Element(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Element) page and scrolled the page to see the properties and methods.
2. To investigate the element Properties header element, we right-clicked on that element and selected ***Inspect*** and selected the <h2> element in the ***Elements*** pane.
3. In the console, we use the shorthand $0 to refer to the element we selected.
4. Next, we checked the className attribute of the element we selected:

$0.className;

1. Next, we looked at the hasAttribute method to see if it has a class:

$0.**hasAttribute**('class');

Just like with the Node interface, every Element has access to every property and method on the MDN [Element(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Element) page.

***Note***: As we discussed earlier, the MDN UI has been updated and no longer uses the highlight-spanned class, so if you try to repeat this demo on your own, you'll see something different in your console:

$0.className;

*// returns ''*

$0.**hasAttribute**('class');

*// returns false*

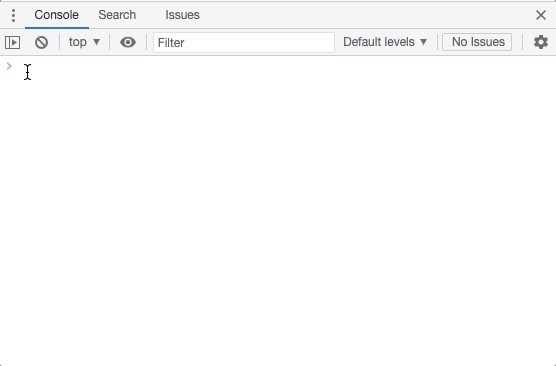
What other attributes can you look for? How about id?

$0.id;

*// returns 'properties'*

$0.**hasAttribute**('id');

*// returns true*



Checking for the id Property of the Element

## A Few Familiar Methods

Do you remember the .getElementsByClassName() method on the document object that we looked at previously? While reviewing the Element interface, you might've noticed that it also has a .getElementsByClassName() method that works just like the .getElementsByClassName() on the document object.

Note that while these work the same, they are actually different methods:

* [Element.getElementsByClassName()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Element/getElementsByClassName)
* [Document.getElementsByClassName()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByClassName)

The Element Interface inherits from the Node Interface, not the Document Interface (yep, there's a Document Interface!).

The good news is that you can use the document object to select an element, then you can call .getElementsByClassName() on that element to receive a list of elements with the class name that are descendents of that specific element!

*// select the DOM element with an ID of "sidebar"*

**const** sidebarElement = document.**getElementById**('sidebar');

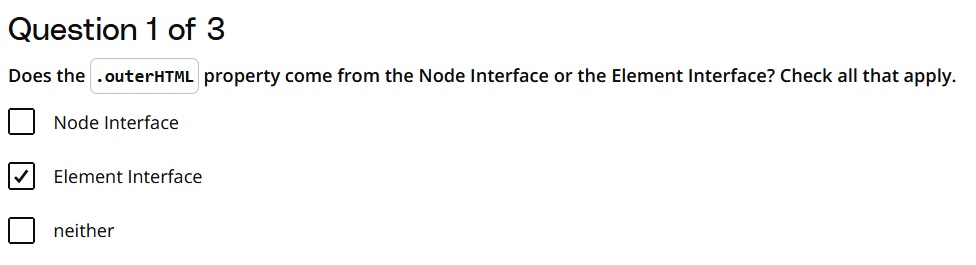
*// search within the "sidebar" element for any elements with a class of "sub-heading"*

**const** subHeadingList = sidebarElement.**getElementsByClassName**('sub-heading');

Which other methods in the Element interface are similar the methods in the Document interface? Follow these links and compare:

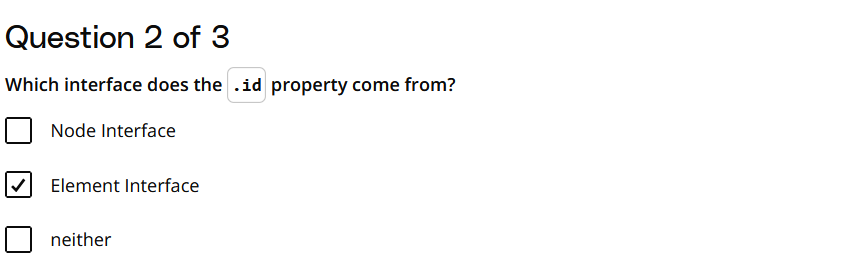
* [MDN: Element Interface(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Element)
* [MDN: Document Interface](https://developer.mozilla.org/en-US/docs/Web/API/Document)

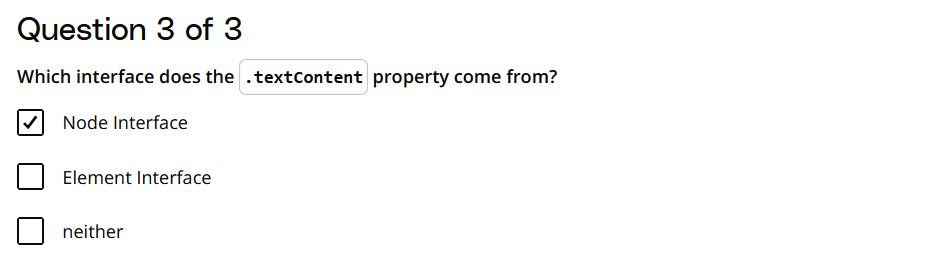
# Quizzes: Nodes, Elements and Interfaces



### Explanation:

* The **Element Interface** represents an HTML or XML element, and it includes properties and methods that are specific to elements in the DOM, such as .outerHTML.
* The **Node Interface** is more general and represents any node in the DOM (like elements, text nodes, comments, etc.), but it does not provide the .outerHTML property.





# More Interfaces

## So Many Interfaces to Use With JavaScript!

Web APIs

# Recap: Nodes, Elements and Interfaces

## Understanding Nodes, Elements, and Interfaces

I hope this was an enlightening lesson on a number of fronts! You learned about interfaces, properties, and methods:

* An **Interface** is like a blueprint
* **Properties** are like bits of information or data
* **Methods** are functionality.

We also looked at a couple of specific interfaces:

* Node Interface
* Element Interface

We saw that both of these interfaces have properties and methods.

We also saw how the **Element** Interface inherits all of the properties and methods from the **Node** Interface.

And then we learned that there are a lot of other interfaces that we can access in JavaScript. Many of these interfaces inherit from other interfaces, which means that they share the properties and methods of their ancestor interfaces.

### Further Research

* [MDN: Node Interface(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Node)
* [MDN: Element Interface(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Element)
* [MDN: Web APIs (opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API) - a list of Web API interfaces

# More Ways To Access Elements

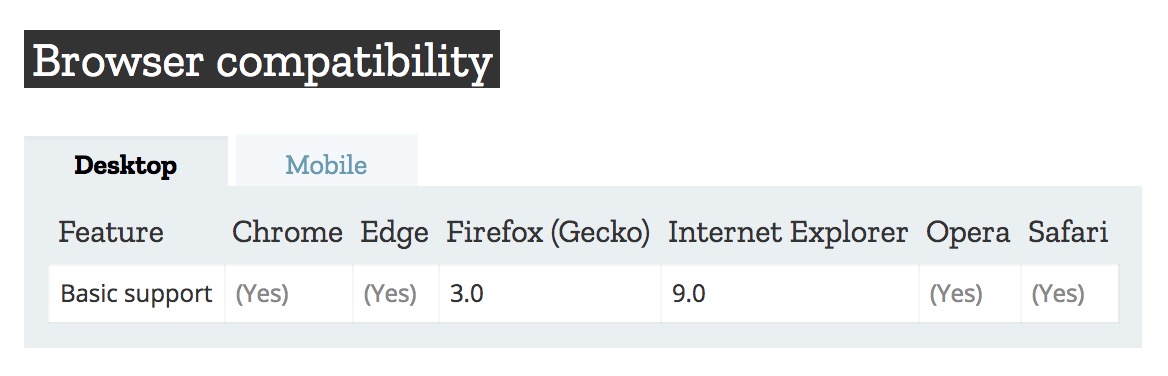
## Quick Recap and a Bit of History

We've been looking at these methods:

* .getElementById()
* .getElementsByClassName()
* .getElementsByTagName()

Now these DOM methods are standardized. However, not all browsers support every standard. They do now, for these three methods, but there are hundreds of other methods with varying levels of support.

That's why almost every method on MDN has a Browser compatibility table that lists when each browser started supporting that specific method.



The Browser compatibility table for the *.getElementsByClassName()* method.

Thankfully, all browsers have pretty much aligned to support the official standard.

However, back in the day, that wasn't the case. You had to write different code to perform the same action in different browsers. Then you had to write code to check which browser you were in to run the correct code for that browser. Let me tell you, it was a bit of a nightmare.

Several JavaScript libraries came along to help mitigate these issues. Let's take a brief look at the [jQuery library(opens in a new tab)](https://jquery.com/).

# querySelector

## Review: Accessing Elements with CSS Selectors

We already reviewed this in a previous section, but let's recap it one more time!

#header {

color: 'red';

}

.header {

color: 'red';

}

header {

color: 'red';

}

Each one of these sets the color to red. The only difference is in the selector; selecting by ID, selecting by class, and selecting by tag. Got it? Good!

You've already learned the DOM methods to select by ID, class, and tag, too:

* document.getElementById()
* document.getElementsByClassName()
* document.getElementsByTagName()

Three different methods that do almost the exact same thing. Wouldn't it be awesome if there were a way to do element selecting similar to how CSS does it?

Wait for it - there is!

## The querySelector Method

We can use the .querySelector() method to select elements just like we do with CSS. We use the .querySelector() method and pass it a string that's just like a CSS selector:

*// find and return the element with an ID of "header"*

document.**querySelector**('#header');

*// find and return the first element with the class "header"*

document.**querySelector**('.header');

*// find and return the first <header> element*

document.**querySelector**('header');

Check out the .querySelector() method on MDN: [Document.querySelector()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector)

### Demo Recap

In this demo:

1. First, we navigated to the MDN [Document.querySelector()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector) page.
2. We right-clicked on the page and selected ***Inspect*** to open the DevTools console.
3. In the console, selected the h2 elements using the querySelector method using the tag name, and then the class:

document.**querySelector**('h2');

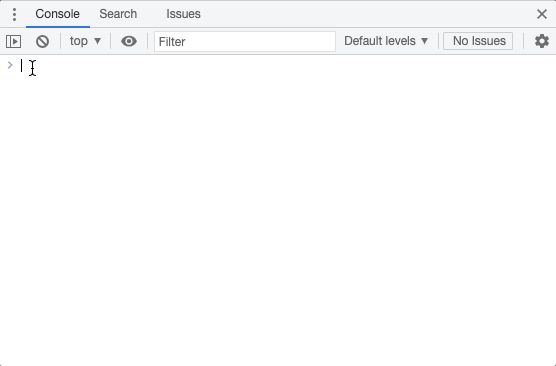
document.**querySelector**('.highlight-spanned');

In both cases we returned the same element!

***Note***: As we discussed earlier, the MDN UI has been updated and no longer uses the highlight-spanned class. If you want to try this on your own (and I would encourage you to do so!), try using the document-toc-heading class instead:

document.**querySelector**('h2');

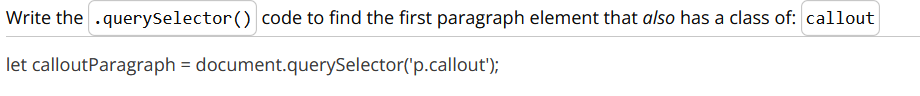
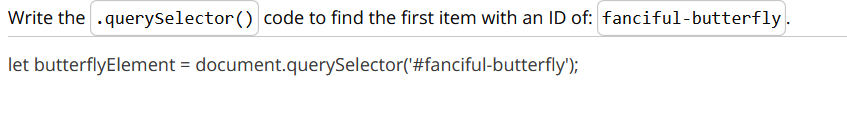
document.**querySelector**('.document-toc-heading');



Two ways to select an *h2* element with *document.querySelector()*

### ⚠️ .querySelector() Returns A Single Element ⚠️

I want to point out one potentially tricky thing - the .querySelector() method only returns one element. This makes sense if you use it to search for an element by ID. However, even though .getElementsByClassName() and .getElementsByTagName() both return a list of multiple elements, using .querySelector() with a class selector or a tag selector will still only return the first item it finds.



# querySelectorAll

## The querySelectorAll Method

The .querySelector() method returns only one element from the DOM (if it exists). However, there are definitely times when you will want to get a list of all elements with a certain class or all of one type of element (e.g. all <tr> tags). We can use the .querySelectorAll() method to do this!

*// find and return a list of elements with the class "header"*

document.**querySelectorAll**('.header');

*// find and return a list of <header> elements*

document.**querySelectorAll**('header');

Here's the .querySelectorAll() method on MDN: [Document.querySelectorAll()](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll)

### Demo Recap

In this demo:

1. First, we navigated to the MDN [Document.querySelectorAll()(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll) page.
2. We right-clicked on the page and selected ***Inspect*** to open the DevTools console.
3. In the console, selected the h2 elements using the querySelectorAll method using the tag name, and then the class:

document.**querySelectorAll**('h2');

1. We stored the output in a variable named listOfElements.

**const** listOfElements = document.**querySelectorAll**('h2');

1. We created a loop to iterate through the NodeList:

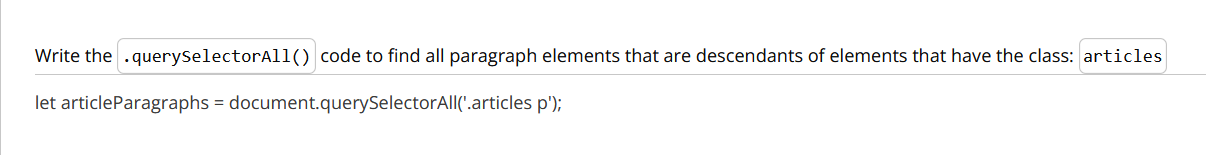
**for** (**let** i = 0; i < listOfElements.length; i++){

**console**.**log**('i is ' + i );

**console**.**log**(listOfElements[i]);

}

It's important to remember that .querySelectorAll() returns a ***NodeList*** -- not an array so listOfElements won't have access to the array methods we are used to. But NodeList has its own methods. You can see them on the MDN [NodeList(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/NodeList) page.



# Recap: querySelector and querySelectorAll

## Recap

In this section, we took a brief look the history of browser support for standard DOM methods, the rise of the jQuery library, and how jQuery's success brought about new DOM methods. The new DOM methods we looked at are

* .querySelector() - returns a single element
* .querySelectorAll() - returns a list of elements

*// find and return the element with an ID of "header"*

document.**querySelector**('#header');

*// find and return a list of elements with the class "header"*

document.**querySelectorAll**('.header');

We also took a brief look that the list returned by .querySelectorAll() is a NodeList. We saw that it is possible to loop over a NodeList with either its .forEach() method, or the humble for loop:

**const** allHeaders = document.**querySelectorAll**('header');

**for**(**let** i = 0; i < allHeaders.length; i++){

**console**.**dir**(allHeaders[i]);

}

### Further Research

* [.querySelector() method on MDN(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector)
* [.querySelectorAll() method on MDN(opens in a new tab)](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll)
* [NodeList on MDN](https://developer.mozilla.org/en-US/docs/Web/API/NodeList)

# Lesson Summary

## You've Learned A Lot!!!

In this lesson, we learned all about the DOM (Document Object Model).

We started by learning what the DOM is and how it gets created.

Then we discovered several JavaScript methods for programmatically accessing DOM elements:

* .getElementById()
* .getElementsByClassName()
* .getElementsByTagName()
* .querySelector()
* .querySelectorAll()

Along the way we talked about the web interfaces that facilitate this access, including:

* Element
* Node

Up next, we'll learn new DOM methods that allow us to alter and control page content!