Class 1

html = like bones

css = features

javascript = logic

html = hypertext markup language

* standard for web pages
* elements are the building blocks

<p> is paragraph </p> closing tab

<a> anchor tags

<a href = “link.com”>

for clickable images, image tag must be nested into anchor tabs (anchor tabs make the image clickable)

./ same source from the html page

Free code Camp

<h1> 1st heading: title

<p> paragraph of text

<ol>

<li> #1 on the list

<!-- comment -->

<header>

<footer>

<nav>

<video>

<article>

<section>

<main> helps search engines and others find the main content of page

Insert an image

**<img src = “image”>** the img points to a specific image’s URL using the src attribute

All img must have an alt tag - alt tag needs to be in quotes

do alt tags describe the image?

**<img src= “website” alt = “two thumbs up”>** <!--error screen pops up and shows “two thumbs up” -->

**<a href=“apples.org”> this links will take you to the website “apples.org” </a>** <!-- the website will tell you it will bring you to apples.org

output: This links to apples.org (all the words will be clickable)

anchor

when the link is clicked, user will be directed to apples.org (this is where the anchor is)

the text that appears clickable

**<hr>** makes a line that creates a line (no closing)

internal link

assign a link’s href to a # and the value of the id attribute for the element you want to internally link - then add the same id attribute to the element you are linking to

id = attribute which uniquely describes an element

**<a href=”#contacts”> Contacts</a>** <!--creates an anchor for the external link which will look for “Contacts”-->

anchor will look for the code “contacts”

This portion will be clickable for the user

**<h2 id=”contacts”</h2>** <!--assigns the id of contacts to this specific header -->

the second header text

this portion is defined as “contacts”

when users click contact they’ll be taken to the section of the webpage with the Contacts header element

another example: <a href="#footer">Jump to Bottom</a>

so what appears on the page is “jump to bottom” and it will be directed to the footer portion

**<a href=”website” target=”\_blank”>cat photos </a>** <!--target=”\_blank” opens a new windows tab so that the cat photo can be opened ; cat photos will appear blue and is the hyperlink to open a new window to see the cat photos

<footer id=”footer”>Copyright Cat Photo App</footer>

shows up as: “Copyright Cat Photo App” footer directs where on the page the text will be located ; why do we define id=footer?

target is an anchor tag attribute which specifies where to open the link: “\_blank” means a new tab

href is an anchor tag which contains the URL address of the link: <a href=”website”>link of apples website</a>

Nesting (Put a link into text)

<p>  
Here's a <a target="\_blank" href="http://freecodecamp.org"> link to freecodecamp.org</a> for you to follow.  
</p>

Final output: “Here’s a link to freecodecamp.org for you to follow.”

the <p> gives: “here’s a <a … </a> for you to follow”

the <a…</a> is nested within <p>…</p>

<a target=”\_blank” opens a new window

href=”website”>link to freecdecamp.org</a> directs to the website makes the text clickable ends the anchor (meaning text not clickable) and paragraph text resumes

instead of href=”website”

href=”#” a deadlink ; you don’t know where you want the website to link yet (placeholder)

Turn an image into a deadlink?

<a href="#"><img src="https://bit.ly/fcc-running-cats" alt="Three kittens running towards the camera."></a>

opening the anchor is important ; so are the <> in between

anchor tag which leads to a deadlink

image(img) sourced(src) from this website

if it fails to work this will show up

don’t forget to close the anchor

At the top of your document, you need to tell the browser which version of HTML your page is using. HTML is an evolving language, and is updated regularly. Most major browsers support the latest specification, which is HTML5. However, older web pages may use previous versions of the language.

<!DOCTYPE …>

… is the version of HTML that will be used

Currently we use HTML5 so: <!DOCTYPE html>

html isn’t case sensitive

the rest of your html code needs to be wrapped in html tags.

First line: <!DOCTYPE html>

second line: <html>

closing line: </html>

You can organize more with head and body elements. Information about your page goes under head element while content of the page goes under body.

Metadata elements like: link, meta, title, style usually go inside head element

<!DOCTYPE html>  
<html>  
  <head>  
    <!-- metadata elements -->  
  </head>  
  <body>  
    <!-- page contents -->  
  </body>  
</html>

tells which type of html the code is using

organize the information; this will be at the top of the page?

a comment that metadata pieces can go here

body of the page

comment that page contents can go here

Class 2

<ul> unordered lists (bullet point)

<li> item </li>

<ol> < li> item </li> ordered list

<li> is children which you need for lists

<input type=”text” placeholder=”text fields”>

This will be in grey

No closing tag

<form action:”/url-to-submit-data-to”><form>

Form is parent; input is children

<button type=”submit”> creates a submit button Once you click submit, the data goes to where the action url directs it

Radio buttons: <label> <input type=”radio” name =”on-off”> On </label>

Input id = on and off

<div> container for other elements

Unordered lists: <ul> </li>

Ordered lists <ol> </li>

Create a text field: <input type=”text”> inputs are self-closing

Create a placeholder in the text field (the greyed out words that tell you what to type)

<input type=”text” placeholder=”write your name”>

Create a web form which sends data somewhere:

<form action=”/url-where-you-want-to-submit-form-data”> </form>

Nest within means just put it between <> and </>

Add a submit button:

<button type=”submit> this button submits the form</button> words on the button

Make a field required in a form:

<input type=”text” required>

Create a set of radio buttons (only one answer from multiple options)

-radio buttons are a type of input

-they can be nested within their own label element

-do this by wrapping an input element inside of a label

-all related radio buttons should have the same name attribute to create a radio button group

<label>

<input type=”radio” name =”indoor-outdoor”> Indoor </label>

The text next to the button

Name – specifies which radio buttons you will interact with

It is considered best practice to set a for attribute on the label element, with a value that matches the value of the id attribute of the input element. This allows assistive technologies to create a linked relationship between the label and the child input element. For example:

<label for=”indoor”>

<input id= “indoor” type = “radio”

Name = “indoor-outdoor”> Indoor

</label>

What is a linked relationship?

Create a set of checkboxes:

-checkboxes are a type of input

-they can be nested inside labels

-do this by wrapping an input element inside a label (it will associate checkbox input with label element surrounding it)

-all related checkbox inputs should have the same name attribute

It is considered best practice to explicitly define the relationship between a checkbox inputand its corresponding labelby setting the forattribute on the labelelement to match the idattribute of the associated inputelement.

<label for=”loving”> <input id=”loving” type=”checkbox” name=”personality”> Loving</label>

Class 2

CSS

-

In the <head> section of your html code, import your style sheet.

<head> <link rel=”stylesheet” href=”styles.css> </head>

Selectors allow you to style all elements of a certain type within a style block

h2{ color: blue;}

Classes are reusable styles which can be added to html elements

Rem = relative units (made to scale)

Change the color of text:

<h2 style=”color: blue;”>CatPhotoApp<h/h2>

End inline style declarations with ; this is inline style ; this is only for one specific section

Create a style block (this is for all elements in that section):

<style> </style>

-you can nest a selector for all h2 elements inside the style block

<style>

h2{color: red;} </style>

Use a CSS class to style an element:

-classes are reusable styles that can be added to HTML elements

<style>

.blue-text {color: blue;} </style>

You can name this whatevever

This is the color it actually will be

You can apply a class to an HTML element like this: <h2 class=”blue-text”>CatPhotoApp</h2>

-with CSS style element, all class names start with a period

Font size:

h1 { font-size: 30px;}

p { font-size: 30px} (front tells which sections will have their font changed)

Font options:

h2 {font-family: sans-serif;}

monospace

Import a google font:

Important note – when copying and pasting quotes, you need to re-type the quotes and possibly the letters – they can be parsed differently

<link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet" type="text/css">

Place this before the style heading

Name of Google font

-now you can use lobster in your CSS (Lobster is family name

h2 {font-family: FAMILY\_NAME, GENERIC\_NAME; }

-generic name is default if family name fails (monospace, sans-serif, serif)

-family names are case sensitive ; if there is a space in the name use quotes (“open sans”)

Sizing images:

<style>

.larger-image {width: 500px} </style>

Names the object (references an object name)

For sizing your images, first create your class in your style tag

<style>

.fixed-image {width: 500px; height: auto; }

<img class= “fixed-image” src=<http://www.example.com/picture>/> #image class will not be in the style bracket, but further down the page closer to the source

Style is only for over-writing

So it’s just class = “blue-text”

Class 3

Add borders around your elements:

<style>

.thin-red-border {border-color: red; border-width: 5px; border-style: solid; border-radius:10x}

</style>

<img class="class1 class2"> assign two classes to one object

Add rounded corners:

border-radius:10px/50%

Give a background color to a div:

.green-background {background-color: green;}

<div class="silver-background">

Color hex codes for more nice colors

Nav { background; linear-gradient (60deg, #62eid7, #e16259) color: white

Text-align: center

Margin: auto fills the space

If you just want to style the same things under one class; won’t touch other unordered lists

nav ul { {

display: inline; all lists are in one line

margin: 15px on both sides 15px

padding-bottom: 10px

margin = space outside the border of an element

padding = space inside the border of an element

hover states:

a:hover { color: #ffe06c; font-size; 1.2 rem; font-family: monospace;}

when your mouse hovers, the color changes, font changes, and text changes

width: fit-content

wrap a border around a text-border

you can style all buttons

button { font-family: monospace;}

button:hover { background: #7864ed; transition: 3s}

transition-delay:3s;

GitHub:

cd Desktop/WebCo-Class-1

git add .

git commit -m “your comment”

git push

CSS

In addition to classes, each HTML element can also have an idattribute.

There are several benefits to using idattributes: You can use an idto style a single element and later you'll learn that you can use them to select and modify specific elements with JavaScript.

idattributes should be unique. Browsers won't enforce this, but it is a widely agreed upon best practice. So please don't give more than one element the same idattribute.

Here's an example of how you give your h2element the id of cat-photo-app:

<h2 id="cat-photo-app">

What does action do? Form action =??

One cool thing about idattributes is that, like classes, you can style them using CSS.

However, an idis not reusable and should only be applied to one element. An idalso has a higher specificity (importance) than a class so if both are applied to the same element and have conflicting styles, the styles of the idwill be applied.

Here's an example of how you can take your element with the idattribute of cat-photo-elementand give it the background color of green. In your styleelement:

#cat-photo-element {  
  background-color: green;  
}

Note that inside your styleelement, you always reference classes by putting a .in front of their names. You always reference ids by putting a #in front of their names.

Now let's put our Cat Photo App away for a little while and learn more about styling HTML.

You may have already noticed this, but all HTML elements are essentially little rectangles.



Three important properties control the space that surrounds each HTML element: padding, margin, and border.

An element's padding controls the amount of space between the element's content and its border.

Here, we can see that the blue box and the red box are nested within the yellow box. Note that the red box has more paddingthan the blue box.

When you increase the blue box's padding, it will increase the distance(padding) between the text and the border around it.

.blue-box {

background-color: blue;

color: #fff;

padding: 20px;

An element's margincontrols the amount of space between an element's borderand surrounding elements.

Here, we can see that the blue box and the red box are nested within the yellow box. Note that the red box has a bigger marginthan the blue box, making it appear smaller.

When you increase the blue box's margin, it will increase the distance between its border and surrounding elements.

An element's margincontrols the amount of space between an element's borderand surrounding elements.

If you set an element's marginto a negative value, the element will grow larger.

.blue-box {

background-color: blue;

color: #fff;

padding: 20px;

margin: -15px;

Sometimes you will want to customize an element so that it has different amounts of paddingon each of its sides.

CSS allows you to control the paddingof all four individual sides of an element with the padding-top, padding-right, padding-bottom, and padding-leftproperties.

Sometimes you will want to customize an element so that it has a different marginon each of its sides.

CSS allows you to control the marginof all four individual sides of an element with the margin-top, margin-right, margin-bottom, and margin-leftproperties.

Instead of specifying an element's padding-top, padding-right, padding-bottom, and padding-leftproperties individually, you can specify them all in one line, like this:

padding: 10px 20px 10px 20px;

These four values work like a clock: top, right, bottom, left, and will produce the exact same result as using the side-specific padding instructions.

Margin works the same way; clockwise

margin: 10px 20px 10px 20px;

Class 4

javascript

// make a comment in java (for one line)

/\* \*/ multi line comment

In computer science, data is anything that is meaningful to the computer. JavaScript provides seven different data types which are undefined, null, boolean, string, symbol, number, and object.

For example, computers distinguish between numbers, such as the number 12, and strings, such as "12", "dog", or "123 cats", which are collections of characters. Computers can perform mathematical operations on a number, but not on a string.

Variables allow computers to store and manipulate data in a dynamic fashion. They do this by using a "label" to point to the data rather than using the data itself. Any of the seven data types may be stored in a variable.

Variablesare similar to the x and y variables you use in mathematics, which means they're a simple name to represent the data we want to refer to. Computer variablesdiffer from mathematical variables in that they can store different values at different times.

We tell JavaScript to create or declare a variable by putting the keyword varin front of it, like so:

Var ourName; // creates a variable called ourName (we end statements with ;)

Variablenames can be made up of numbers, letters, and $or \_, but may not contain spaces or start with a number.

In JavaScript, you can store a value in a variable with the assignment operator.

myVariable = 5;

This assigns the Numbervalue 5to myVariable.

Assignment always goes from right to left. Everything to the right of the =operator is resolved before the value is assigned to the variable to the left of the operator.

myVar = 5;  
myNum = myVar;

This assigns 5to myVarand then resolves myVarto 5again and assigns it to myNum.

You can create a new variable and assign it with a value.

When JavaScript variables are declared, they have an initial value of undefined. If you do a mathematical operation on an undefinedvariable your result will be NaNwhich means "Not a Number". If you concatenate a string with an undefinedvariable, you will get a literal string of "undefined".

In JavaScript all variables and function names are case sensitive. This means that capitalization matters.

MYVARis not the same as MyVarnor myvar. It is possible to have multiple distinct variables with the same name but different casing. It is strongly recommended that for the sake of clarity, you do not use this language feature.

Best Practice

Write variable names in JavaScript in *camelCase*. In *camelCase*, multi-word variable names have the first word in lowercase and the first letter of each subsequent word is capitalized.

You can add (+) subtract (-) and multiply (\*) and divide (/)

You can easily increment or add one to a variable with the ++operator.

i++;

is the equivalent of

i = i + 1;

**Note**  
The entire line becomes i++;, eliminating the need for the equal sign.

myVar = 87

myVar ++; // myVar = 88

create decimals by using . (5.7)

The remainder operator %gives the remainder of the division of two numbers.

**Example**

5 % 2 = 1 because  
Math.floor(5 / 2) = 2 (Quotient)  
2 \* 2 = 4  
5 - 4 = 1 (Remainder)

**Usage**  
In mathematics, a number can be checked to be even or odd by checking the remainder of the division of the number by 2.

17 % 2 = 1 (17 is Odd)  
48 % 2 = 0 (48 is Even)

**Note**  
The remainder operator is sometimes incorrectly referred to as the "modulus" operator. It is very similar to modulus, but does not work properly with negative numbers.

11 % 3 = 2

## **Compound Assignment With Augmented Addition**

Remember that everything to the right of the equals sign is evaluated first, so we can say:

myVar = myVar + 5;

to add 5to myVar. Since this is such a common pattern, there are operators which do both a mathematical operation and assignment in one step.

One such operator is the +=operator.

var myVar = 1;  
myVar += 5;  
console.log(myVar); // Returns 6

-= # (substracts # from the number in front)

\*= # (multplies …)

/= (divides…)

Previously we have used the code

var myName = "your name";

"your name"is called a string literal. It is a string because it is a series of zero or more characters enclosed in single or double quotes.

When you are defining a string you must start and end with a single or double quote. What happens when you need a literal quote: "or 'inside of your string?

In JavaScript, you can escape a quote from considering it as an end of string quote by placing a backslash (\) in front of the quote.

var sampleStr = "Alan said, \"Peter is learning JavaScript\".";

This signals to JavaScript that the following quote is not the end of the string, but should instead appear inside the string. So if you were to print this to the console, you would get:

Alan said, "Peter is learning JavaScript".

var myStr = "I am a \"double quoted\" string inside \"double quotes\".";

I am a "double quoted" string inside "double quotes".

String values in JavaScript may be written with single or double quotes, as long as you start and end with the same type of quote. Unlike some other programming languages, single and double quotes work the same in JavaScript.

doubleQuoteStr = "This is a string";  
singleQuoteStr = 'This is also a string';

The reason why you might want to use one type of quote over the other is if you want to use both in a string. This might happen if you want to save a conversation in a string and have the conversation in quotes. Another use for it would be saving an <a>tag with various attributes in quotes, all within a string.

conversation = 'Finn exclaims to Jake, "Algebraic!"';

However, this becomes a problem if you need to use the outermost quotes within it. Remember, a string has the same kind of quote at the beginning and end. But if you have that same quote somewhere in the middle, the string will stop early and throw an error.

goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?"';  
badStr = 'Finn responds, "Let's go!"'; // Throws an error

In the goodStr above, you can use both quotes safely by using the backslash \as an escape character.

**Note**  
The backslash \should not be be confused with the forward slash /. They do not do the same thing.

var myStr = "<a href=\"http://www.example.com\" target=\"\_blank\">Link</a>";

What does <a> do?

Quotes are not the only characters that can be *escaped* inside a string. There are two reasons to use escaping characters: First is to allow you to use characters you might not otherwise be able to type out, such as a backspace. Second is to allow you to represent multiple quotes in a string without JavaScript misinterpreting what you mean. We learned this in the previous challenge.

| **Code** | **Output** |
| --- | --- |
| \' | single quote |
| \" | double quote |
| \\ | backslash |
| \n | newline |
| \r | carriage return |
| \t | tab |
| \b | backspace |
| \f | form feed |

*Note that the backslash itself must be escaped in order to display as a backslash.*

Assign the following three lines of text into the single variable myStrusing escape sequences.

FirstLine  
    \SecondLine  
ThirdLine

You will need to use escape sequences to insert special characters correctly. You will also need to follow the spacing as it looks above, with no spaces between escape sequences or words.

Here is the text with the escape sequences written out.

FirstLinenewlinetabbackslashSecondLinenewlineThirdLine

var myStr ='FirstLine \n \t SecondLine \n ThirdLine'

var myStr ='FirstLine\n\tSecondLine\nThirdLine'

In JavaScript, when the +operator is used with a Stringvalue, it is called the concatenation operator. You can build a new string out of other strings by concatenating them together.

**Example**

'My name is Alan,' + ' I concatenate.'

**Note**  
Watch out for spaces. Concatenation does not add spaces between concatenated strings, so you'll need to add them yourself.

var myStr ="This is the start." + "This is the end.";

We can also use the +=operator to concatenate a string onto the end of an existing string variable. This can be very helpful to break a long string over several lines.

**Note**  
Watch out for spaces. Concatenation does not add spaces between concatenated strings, so you'll need to add them yourself.

var myStr = "This is the first sentence."

myStr += " This is the second sentence." Concatenation

Sometimes you will need to build a string, [Mad Libs](https://en.wikipedia.org/wiki/Mad_Libs) style. By using the concatenation operator (+), you can insert one or more variables into a string you're building.

var ourName = "freeCodeCamp";

var ourStr = "Hello, our name is " + ourName + ", how are you?";

Just as we can build a string over multiple lines out of string literals, we can also append variables to a string using the plus equals (+=) operator.

var someAdjective = "peaches";

var myStr = "Learning to code is ";

myStr += someAdjective

You can find the length of a Stringvalue by writing .lengthafter the string variable or string literal.

"Alan Peter".length; // 10

For example, if we created a variable var firstName = "Charles", we could find out how long the string "Charles"is by using the firstName.lengthproperty.

var firstNameLength = 0;

var firstName = "Ada";

firstNameLength = firstName.length;

Bracket notationis a way to get a character at a specific indexwithin a string.

Most modern programming languages, like JavaScript, don't start counting at 1 like humans do. They start at 0. This is referred to as Zero-based indexing.

For example, the character at index 0 in the word "Charles" is "C". So if var firstName = "Charles", you can get the value of the first letter of the string by using firstName[0].

Example 1

var firstLetterOfFirstName = "";

var firstName = "Ada";

firstLetterOfFirstName = firstName[0];

example 2

var firstLetterOfLastName = "";

var lastName = "Lovelace";

firstLetterOfLastName = lastName.length

firstLetterOfLastName = lastName[0]

In JavaScript, Stringvalues are immutable, which means that they cannot be altered once created.

For example, the following code:

var myStr = "Bob";  
myStr[0] = "J";

cannot change the value of myStrto "Job", because the contents of myStrcannot be altered. Note that this does not mean that myStrcannot be changed, just **that the individual characters of a string literal cannot be changed**. The only way to change myStrwould be to assign it with a new string, like this:

var myStr = "Bob";  
myStr = "Job";

You can also use bracket notation to get the character at other positions within a string.

Remember that computers start counting at 0, so the first character is actually the zeroth character.

var firstName = "Ada";

var secondLetterOfFirstName = firstName[1];

In order to get the last letter of a string, you can subtract one from the string's length.

For example, if var firstName = "Charles", you can get the value of the last letter of the string by using firstName[firstName.length - 1].

var lastName = "Lovelace";

var lastLetterOfLastName = lastName[lastName.length -1];

You can use the same principle we just used to retrieve the last character in a string to retrieve the Nth-to-last character.

For example, you can get the value of the third-to-last letter of the var firstName = "Charles"string by using firstName[firstName.length - 3]

var firstName = "Ada";

var thirdToLastLetterOfFirstName = firstName[firstName.length - 3];

We will now use our knowledge of strings to build a "[Mad Libs](https://en.wikipedia.org/wiki/Mad_Libs)" style word game we're calling "Word Blanks". You will create an (optionally humorous) "Fill in the Blanks" style sentence.

In a "Mad Libs" game, you are provided sentences with some missing words, like nouns, verbs, adjectives and adverbs. You then fill in the missing pieces with words of your choice in a way that the completed sentence makes sense.

Consider this sentence - "It was really **\_\_\_\_**, and we **\_\_\_\_** ourselves **\_\_\_\_**". This sentence has three missing pieces- an adjective, a verb and an adverb, and we can add words of our choice to complete it. We can then assign the completed sentence to a variable as follows:

var sentence = "It was really" + "hot" + ", and we" + "laughed" + "ourselves" + "silly.";

myNoun = "dog"

myAdjective = "big"

myVerb = "ran"

myAdverb = "quickly"

result = "My " + myNoun + " was " + myAdjective + " and " + myVerb + " very " + myAdverb + "."

With JavaScript arrayvariables, we can store several pieces of data in one place.

You start an array declaration with an opening square bracket, end it with a closing square bracket, and put a comma between each entry, like this:

var sandwich = ["peanut butter", "jelly", "bread"].

var myArray = ["bark", 7]; (numbers don’t get quotes)

You can also nest arrays within other arrays, like this: [["Bulls", 23], ["White Sox", 45]]. This is also called a Multi-dimensional Array.

var myArray = [["woof", "bark"], ["meow", "rawr"]]

We can access the data inside arrays using indexes.

Array indexes are written in the same bracket notation that strings use, except that instead of specifying a character, they are specifying an entry in the array. Like strings, arrays use zero-based indexing, so the first element in an array is element 0.

**Example**

var array = [50,60,70];  
array[0]; // equals 50  
var data = array[1]; // equals 60

**Note**  
There shouldn't be any spaces between the array name and the square brackets, like array [0]. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

var myArray = [50,60,70];

var myData = myArray[0]

Unlike strings, the entries of arrays are mutable and can be changed freely.

**Example**

var ourArray = [50,40,30];  
ourArray[0] = 15; // equals [15,40,30]

**Note**  
There shouldn't be any spaces between the array name and the square brackets, like array [0]. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

myArray[0] = 45

One way to think of a multi-dimensional array, is as an array of arrays. When you use brackets to access your array, the first set of brackets refers to the entries in the outer-most (the first level) array, and each additional pair of brackets refers to the next level of entries inside.

**Example**

var arr = [  
  [1,2,3],  
  [4,5,6],  
  [7,8,9],  
  [[10,11,12], 13, 14]  
];  
arr[3]; // equals [[10,11,12], 13, 14]  
arr[3][0]; // equals [10,11,12]  
arr[3][0][1]; // equals 11

var myArray = [[1,2,3], [4,5,6], [7,8,9], [[10,11,12], 13, 14]];

var myData = myArray[2][1]; (myData = 8)

An easy way to append data to the end of an array is via the push()function.

.push()takes one or more parameters and "pushes" them onto the end of the array.

var arr = [1,2,3];  
arr.push(4);  
// arr is now [1,2,3,4]

Push ["dog", 3]onto the end of the myArrayvariable.

var myArray = [["John", 23], ["cat", 2]];

myArray.push(["dog",3])

myArrayshould now equal [["John", 23], ["cat", 2], ["dog", 3]]

Another way to change the data in an array is with the .pop()function.

.pop()is used to "pop" a value **off of the end of an array**. We can store this "popped off" value by assigning it to a variable. In other words, .pop()removes the last element from an array and returns that element.

Any type of entry can be "popped" off of an array - numbers, strings, even nested arrays.

var threeArr = [1, 4, 6];  
var oneDown = threeArr.pop();  
console.log(oneDown); // Returns 6  
console.log(threeArr); // Returns [1, 4]

var myArray = [["John", 23], ["cat", 2]];

var removedFromMyArray = myArray.pop()

emovedFromMyArrayshould only contain ["cat", 2].

pop()always removes the last element of an array. What if you want to remove the first?

That's where .shift()comes in. It works just like .pop(), except it removes the first element instead of the last.

var myArray = [["John", 23], ["dog", 3]];

var removedFromMyArray = myArray.shift()

removedFromMyArrayshould contain ["John", 23].

Not only can you shiftelements off of the beginning of an array, you can also unshiftelements to the beginning of an array i.e. add elements in front of the array.

.unshift()works exactly like .push(), but instead of adding the element at the end of the array, unshift()adds the element at the beginning of the array.

var myArray = [["John", 23], ["dog", 3]];

myArray.shift();

myArray.unshift(["Paul",35])

myArrayshould now have [["Paul", 35], ["dog", 3]].

Create a shopping list in the variable myList. The list should be a multi-dimensional array containing several sub-arrays.

The first element in each sub-array should contain a string with the name of the item. The second element should be a number representing the quantity i.e.

["Chocolate Bar", 15]

There should be at least 5 sub-arrays in the list.

var myList = [["popcorn", 3],["apples", 8],["watermelon", 1],["soap",5],["muffins",7]] ;

In JavaScript, we can divide up our code into reusable parts called functions.

Here's an example of a function:

function functionName() {  
  console.log("Hello World");  
}

You can call or invoke this function by using its name followed by parentheses, like this:

functionName();

Each time the function is called it will print out the message "Hello World"on the dev console. All of the code between the curly braces will be executed every time the function is called.

function resusableFunction () {console.log("Hi World");}

function ourReusableFunction() {

console.log("Heyya, World");

}

ourReusableFunction();

ISSUES WITH THIS ONE ^

// Example

function ourReusableFunction() {

console.log("Heyya, World");

}

ourReusableFunction();

// Only change code below this line

function reusableFunction() {

console.log("Hi World");

}

reusableFunction();

Parameters are variables that act as placeholders for the values that are to be input to a function when it is called. When a function is defined, it is typically defined along with one or more parameters. The actual values that are input (or "passed") into a function when it is called are known as arguments.

Here is a function with two parameters, param1and param2:

function testFun(param1, param2) {  
  console.log(param1, param2);  
}

Then we can call testFun:

testFun("Hello", "World");

We have passed two arguments, "Hello"and "World". Inside the function, param1will equal "Hello" and param2will equal "World". Note that you could call testFunagain with different arguments and the parameters would take on the value of the new arguments.

1. Create a function called functionWithArgsthat accepts two arguments and outputs their sum to the dev console.
2. Call the function with two numbers as arguments.

function functionWithArgs (a, b) {console.log(a+b);}

functionWithArgs(1,2);

functionWithArgs(7,9);

In JavaScript, scope refers to the visibility of variables. Variables which are defined outside of a function block have Global scope. This means, they can be seen everywhere in your JavaScript code.

Variables which are used without the varkeyword are automatically created in the globalscope. This can create unintended consequences elsewhere in your code or when running a function again. You should always declare your variables with var.

Using var, declare a globalvariable myGlobaloutside of any function. Initialize it with a value of 10.

Inside function fun1, assign 5to oopsGlobal**without** using the varkeyword.

// Declare your variable here

var myGlobal = 10

function fun1() {

oopsGlobal = 5

// Assign 5 to oopsGlobal Here

}

// Only change code above this line

function fun2() {

var output = "";

if (typeof myGlobal != "undefined") {

output += "myGlobal: " + myGlobal;

}

if (typeof oopsGlobal != "undefined") {

output += " oopsGlobal: " + oopsGlobal;

}

console.log(output);

}

Variables which are declared **within a function**, as well as the function parameters have **local scope**. That means, they are **only visible within that function**.

Here is a function myTestwith a local variable called loc.

function myTest() {  
  var loc = "foo";  
  console.log(loc);  
}  
myTest(); // logs "foo"  
console.log(loc); // loc is not defined

locis not defined outside of the function.

Declare a local variable myVarinside myLocalScope. Run the tests and then follow the instructions commented out in the editor.

function myLocalScope() {

'use strict'; // you shouldn't need to edit this line

var myVar = "woof"

console.log(myVar);

}

myLocalScope();

// Run and check the console

// myVar is not defined outside of myLocalScope

// Now remove the console log line to pass the test

It is possible to have both local and global variables with the same name. When you do this, the localvariable takes precedence over the globalvariable.

In this example:

var someVar = "Hat";  
function myFun() {  
  var someVar = "Head";  
  return someVar;  
}

The function myFunwill return "Head"because the localversion of the variable is present.

Add a local variable to myOutfitfunction to override the value of outerWearwith "sweater".

// Setup

var outerWear = "T-Shirt";

function myOutfit() {

var outerWear = "sweater"

// Only change code below this line

// Only change code above this line

return outerWear;

}

myOutfit();

return means print?

We can pass values into a function with arguments. You can use a returnstatement to send a value back out of a function.

**Example**

function plusThree(num) {  
  return num + 3;  
}  
var answer = plusThree(5); // 8

plusThreetakes an argument for numand returns a value equal to num + 3.

Create a function timesFivethat accepts one argument, multiplies it by 5, and returns the new value. See the last line in the editor for an example of how you can test your timesFivefunction.

function timesFive(num) {return num\*5;}

timesFive(5);

timesFive(2);

timesFive(0);

A function can include the returnstatement but it does not have to. In the case that the function doesn't have a returnstatement, when you call it, the function processes the inner code but the returned value is undefined.

**Example**

var sum = 0;  
function addSum(num) {  
  sum = sum + num;  
}  
var returnedValue = addSum(3); // sum will be modified but returned value is undefined

addSumis a function without a returnstatement. The function will change the global sumvariable but the returned value of the function is undefined

// Example

var sum = 0;

function addThree() {

sum = sum + 3;

}

// Only change code below this line

function addFive(num) {sum = sum + 5;}

// Only change code above this line

var returnedValue = addFive();

If you'll recall from our discussion of [Storing Values with the Assignment Operator](https://learn.freecodecamp.org/javascript-algorithms-and-data-structures/basic-javascript/storing-values-with-the-assignment-operator), everything to the right of the equal sign is resolved before the value is assigned. This means we can take the return value of a function and assign it to a variable.

Assume we have pre-defined a function sumwhich adds two numbers together, then:

ourSum = sum(5, 12);

will call sumfunction, which returns a value of 17and assigns it to ourSumvariable.

Call the processArgfunction with an argument of 7and assign its return value to the variable processed.

// Example

var changed = 0;

function change(num) {

return (num + 5) / 3;

}

changed = change(10);

// Setup

var processed = 0;

function processArg(num) {

return (num + 3) / 5;

}

// Only change code below this line

processed = processArg(7);

In Computer Science a queue is an abstract Data Structure where items are kept in order. New items can be added at the back of the queueand old items are taken off from the front of the queue.

Write a function nextInLinewhich takes an array (arr) and a number (item) as arguments.

Add the number to the end of the array, then remove the first element of the array.

The nextInLinefunction should then return the element that was removed.

function nextInLine(arr, item) {

nextInLine.push(item) = nextInLine(); // adds the parameter item to the end of the array in the function nextInLine

nextInLine.shift(arr) = nextInLine(); // moves the parameter arr to the front of the array in the function nextInLine

// Your code here

return nextInLine; // Change this line

}

Arr.push(item)

Output = arr.shift()

Return output

Class 5

Another data type is the Boolean. Booleansmay only be one of two values: trueor false. They are basically little on-off switches, where trueis "on" and falseis "off." These two states are mutually exclusive.

**Note**  
Booleanvalues are never written with quotes. The strings"true"and "false"are not Booleanand have no special meaning in JavaScript.

Modify the welcomeToBooleansfunction so that it returns trueinstead of falsewhen the run button is clicked.

function welcomeToBooleans() {

// Only change code below this line.

return true; // Change this line

// Only change code above this line.

}

Ifstatements are used to make decisions in code. The keyword iftells JavaScript to execute the code in the curly braces under certain conditions, defined in the parentheses. These conditions are known as Booleanconditions and they may only be trueor false.

When the condition evaluates to true, the program executes the statement inside the curly braces. When the Boolean condition evaluates to false, the statement inside the curly braces will not execute.

**Pseudocode**

if (*condition is true*) {  
  *statement is executed*  
}

**Example**

function test (myCondition) {  
  if (myCondition) {  
     return "It was true";  
  }  
  return "It was false";  
}  
test(true); // returns "It was true"  
test(false); // returns "It was false"

When testis called with a value of true, the ifstatement evaluates myConditionto see if it is trueor not. Since it is true, the function returns "It was true". When we call testwith a value of false, myConditionis not trueand the statement in the curly braces is not executed and the function returns "It was false".

Create an ifstatement inside the function to return "Yes, that was true"if the parameter wasThatTrueis trueand return "No, that was false"otherwise.

// Setup

function trueOrFalse(wasThatTrue) {

// Only change code below this line.

if (wasThatTrue) { return "Yes, that was true";}

return "No, that was false";

// Only change code above this line.

}

// Change this value to test

trueOrFalse(true);

trueOrFalse(false)

There are many Comparison Operators in JavaScript. All of these operators return a boolean trueor falsevalue.

The most basic operator is the equality operator ==. The equality operator compares two values and returns trueif they're equivalent or falseif they are not. Note that equality is different from assignment (=), which assigns the value at the right of the operator to a variable in the left.

function equalityTest(myVal) {  
  if (myVal == 10) {  
     return "Equal";  
  }  
  return "Not Equal";  
}

If myValis equal to 10, the equality operator returns true, so the code in the curly braces will execute, and the function will return "Equal". Otherwise, the function will return "Not Equal".

In order for JavaScript to compare two different data types(for example, numbersand strings), it must convert one type to another. This is known as "Type Coercion". Once it does, however, it can compare terms as follows:

1 == 1 // true  
1 == 2 // false  
1 == '1' // true  
"3" == 3 // true

Add the equality operatorto the indicated line so that the function will return "Equal" when valis equivalent to 12

function testEqual(val) {

if (val == 12) { // Change this line

return "Equal";

}

return "Not Equal";

}

// Change this value to test

testEqual(10); not equal

testEqual(12); equal

testEqual("12") equal

Strict equality (===) is the counterpart to the equality operator (==). However, unlike the equality operator, which attempts to convert both values being compared to a common type, the strict equality operator does not perform a type conversion.

If the values being compared have different types, they are considered unequal, and the strict equality operator will return false.

**Examples**

3 === 3 // true  
3 === '3' // false

In the second example, 3is a Numbertype and '3'is a Stringtype.

Use the strict equality operator in the ifstatement so the function will return "Equal" when valis strictly equal to 7

// Setup

function testStrict(val) {

if (val===7) { // Change this line

return "Equal";

}

return "Not Equal";

}

// Change this value to test

testStrict(10); not equal

testStrict(7); equal

testStrict("7"); not equal

In the last two challenges, we learned about the equality operator (==) and the strict equality operator (===). Let's do a quick review and practice using these operators some more.

If the values being compared are not of the same type, the equality operator will perform a type conversion, and then evaluate the values. However, the strict equality operator will compare both the data type and value as-is, without converting one type to the other.

**Examples**

3 == '3' // returns true because JavaScript performs type conversion from string to number  
3 === '3' // returns false because the types are different and type conversion is not performed

**Note**  
In JavaScript, you can determine the type of a variable or a value with the typeofoperator, as follows:

typeof 3 // returns 'number'  
typeof '3' // returns 'string'

The compareEqualityfunction in the editor compares two values using the equality operator. Modify the function so that it returns "Equal" only when the values are strictly equal.

// Setup

function compareEquality(a, b) {

if (a === b) { // Change this line

return "Equal";

}

return "Not Equal";

}

// Change this value to test

compareEquality(10, "10");

The inequality operator (!=) is the opposite of the equality operator. It means "Not Equal" and returns falsewhere equality would return trueand vice versa. Like the equality operator, the inequality operator will convert data types of values while comparing.

**Examples**

1 != 2 // true  
1 != "1" // false  
1 != '1' // false  
1 != true // false  
0 != false // false

Add the inequality operator !=in the ifstatement so that the function will return "Not Equal" when valis not equivalent to 99

// Setup

function testNotEqual(val) {

if (val != 99) { // Change this line

return "Not Equal";

}

return "Equal";

}

// Change this value to test

testNotEqual(99);

testNotEqual("99");

testNotEqual(12);

testNotEqual("12");

testNotEqual("bob");

The strict inequality operator (!==) is the logical opposite of the strict equality operator. It means "Strictly Not Equal" and returns falsewhere strict equality would return trueand vice versa. Strict inequality will not convert data types.

**Examples**

3 !== 3 // false  
3 !== '3' // true  
4 !== 3 // true

Add the strict inequality operatorto the ifstatement so the function will return "Not Equal" when valis not strictly equal to 17

// Setup

function testStrictNotEqual(val) {

// Only Change Code Below this Line

if (val !== 17) {

// Only Change Code Above this Line

return "Not Equal";

}

return "Equal";

}

// Change this value to test

testStrictNotEqual(17);

testStrictNotEqual("17");

testStrictNotEqual(12);

testStrictNotEqual("bob");

The greater than operator (>) compares the values of two numbers. If the number to the left is greater than the number to the right, it returns true. Otherwise, it returns false.

Like the equality operator, greater than operator will convert data types of values while comparing.

**Examples**

5 > 3 // true  
7 > '3' // true  
2 > 3 // false  
'1' > 9 // false

Add the greater thanoperator to the indicated lines so that the return statements make sense.

function testGreaterThan(val) {

if (val > 100) { // Change this line

return "Over 100";

}

if (val > 10) { // Change this line

return "Over 10";

}

return "10 or Under";

}

// Change this value to test

testGreaterThan(0);

testGreaterThan(10);

testGreaterThan(11);

testGreaterThan(99);

The greater than or equal tooperator (>=) compares the values of two numbers. If the number to the left is greater than or equal to the number to the right, it returns true. Otherwise, it returns false.

Like the equality operator, greater than or equal tooperator will convert data types while comparing.

**Examples**

6 >= 6 // true  
7 >= '3' // true  
2 >= 3 // false  
'7' >= 9 // false

Add the greater than or equal tooperator to the indicated lines so that the return statements make sense.

function testGreaterOrEqual(val) {

if (val >= 20) { // Change this line

return "20 or Over";

}

if (val >= 10) { // Change this line

return "10 or Over";

}

return "Less than 10";

}

// Change this value to test

testGreaterOrEqual(10);

The less than operator (<) compares the values of two numbers. If the number to the left is less than the number to the right, it returns true. Otherwise, it returns false. Like the equality operator, less than operator converts data types while comparing.

**Examples**

2 < 5 // true  
'3' < 7 // true  
5 < 5 // false  
3 < 2 // false  
'8' < 4 // false

Add the less thanoperator to the indicated lines so that the return statements make sense.

function testLessThan(val) {

if (val < 25) { // Change this line

return "Under 25";

}

if (val <55) { // Change this line

return "Under 55";

}

return "55 or Over";

}

// Change this value to test

testLessThan(10);

less than or equal operator <=

Sometimes you will need to test more than one thing at a time. The logical and operator (&&) returns trueif and only if the operands to the left and right of it are true.

The same effect could be achieved by nesting an if statement inside another if:

if (num > 5) {  
  if (num < 10) {  
    return "Yes";  
  }  
}  
return "No";

will only return "Yes" if numis greater than 5and less than 10. The same logic can be written as:

if (num > 5 && num < 10) {  
  return "Yes";  
}  
return "No";

Combine the two if statements into one statement which will return "Yes"if valis less than or equal to 50and greater than or equal to 25. Otherwise, will return "No".

function testLogicalAnd(val) {

// Only change code below this line

if (val <= 50 && val >= 25)

{ return "Yes"; }

// Only change code above this line

return "No";

}

// Change this value to test

testLogicalAnd(10);

The logical or operator (||) returns trueif either of the operands is true. Otherwise, it returns false.

The logical or operator is composed of two pipe symbols (|). This can typically be found between your Backspace and Enter keys.

The pattern below should look familiar from prior waypoints:

if (num > 10) {  
  return "No";  
}  
if (num < 5) {  
  return "No";  
}  
return "Yes";

will return "Yes" only if numis between 5and 10(5 and 10 included). The same logic can be written as:

if (num > 10 || num < 5) {  
  return "No";  
}  
return "Yes";

Combine the two ifstatements into one statement which returns "Outside"if valis not between 10and 20, inclusive. Otherwise, return "Inside".

function testLogicalOr(val) {

// Only change code below this line

if (val < 10 || val > 20

) {

return "Outside";

}

// Only change code above this line

return "Inside";

}

// Change this value to test

testLogicalOr(15);

Class 5

Input -🡪 function 🡪 output

Input = argument

For function timewarp

Input = 61 (sec)

Output = [0,1,1] use return

When a condition for an ifstatement is true, the block of code following it is executed. What about when that condition is false? Normally nothing would happen. With an elsestatement, an alternate block of code can be executed.

if (num > 10) {  
  return "Bigger than 10";  
} else {  
  return "10 or Less";  
}

Combine the if statements into a single if/elsestatement.

function testElse(val) {

var result = "";

// Only change code below this line

if (val > 5) {

result = "Bigger than 5";

}

else {

result = "5 or Smaller";

}

// Only change code above this line

return result;

}

// Change this value to test

testElse(4);

If you have multiple conditions that need to be addressed, you can chain ifstatements together with else ifstatements.

if (num > 15) {  
  return "Bigger than 15";  
} else if (num < 5) {  
  return "Smaller than 5";  
} else {  
  return "Between 5 and 15";  
}

Convert the logic to use else ifstatements.

function testElseIf(val) {

if (val > 10) {

return "Greater than 10";

}

else if (val < 5) {

return "Smaller than 5";

}

else {

return "Between 5 and 10";

}

}

// Change this value to test

testElseIf(0);

Order is important in if, else ifstatements.

The function is executed from top to bottom so you will want to be careful of what statement comes first.

Take these two functions as an example.

Here's the first:

function foo(x) {  
  if (x < 1) {  
    return "Less than one";  
  } else if (x < 2) {  
    return "Less than two";  
  } else {  
    return "Greater than or equal to two";  
  }  
}

And the second just switches the order of the statements:

function bar(x) {  
  if (x < 2) {  
    return "Less than two";  
  } else if (x < 1) {  
    return "Less than one";  
  } else {  
    return "Greater than or equal to two";  
  }  
}

While these two functions look nearly identical if we pass a number to both we get different outputs.

foo(0) // "Less than one"  
bar(0) // "Less than two"

Change the order of logic in the function so that it will return the correct statements in all cases.

function orderMyLogic(val) {

if (val < 5) {

return "Less than 5";

} else if (val < 10) {

return "Less than 10";

} else {

return "Greater than or equal to 10";

}

}

// Change this value to test

orderMyLogic(7);

if/elsestatements can be chained together for complex logic. Here is pseudocode of multiple chained if/ else ifstatements:

if (condition1) {  
  statement1  
} else if (condition2) {  
  statement2  
} else if (condition3) {  
  statement3  
. . .  
} else {  
  statementN  
}

Write chained if/else ifstatements to fulfill the following conditions:

num < 5- return "Tiny"  
num < 10- return "Small"  
num < 15- return "Medium"  
num < 20- return "Large"  
num >= 20- return "Huge"

function testSize(num) {

// Only change code below this line

if (num < 5) { return "Tiny"}

else if (num < 10) {return "Small"}

else if (num < 15) {return "Medium"}

else if (num < 20) {return "Large"}

else if (num >= 20) {return "Huge"}

return "Change Me";

// Only change code above this line

}

// Change this value to test

testSize(7);

In the game of [golf](https://en.wikipedia.org/wiki/Golf) each hole has a parmeaning the average number of strokesa golfer is expected to make in order to sink the ball in a hole to complete the play. Depending on how far above or below paryour strokesare, there is a different nickname.

Your function will be passed parand strokesarguments. Return the correct string according to this table which lists the strokes in order of priority; top (highest) to bottom (lowest):

| **Strokes** | **Return** |
| --- | --- |
| 1 | "Hole-in-one!" |
| <= par - 2 | "Eagle" |
| par - 1 | "Birdie" |
| par | "Par" |
| par + 1 | "Bogey" |
| par + 2 | "Double Bogey" |
| >= par + 3 | "Go Home!" |

parand strokeswill always be numeric and positive. We have added an array of all the names for your convenience.

var names = ["Hole-in-one!", "Eagle", "Birdie", "Par", "Bogey", "Double Bogey", "Go Home!"];

function golfScore(par, strokes) {

// Only change code below this line

if (strokes == 1) // use == when comparing not = otherwise it will assign the value 1

return "Hole-in-one!";

if (strokes <= par-2)

return "Eagle";

if (strokes <= par-1)

return "Birdie";

if (strokes <= par)

return "Par";

if (strokes <= par+1)

return "Bogey";

if (strokes <= par+2)

return "Double Bogey";

if (strokes >= par+2)

return "Go Home!"

else

return "Change Me";

// Only change code above this line

}

// Change these values to test

golfScore(5, 4);

If you have many options to choose from, use a switchstatement. A switchstatement tests a value and can have many casestatements which define various possible values. Statements are executed from the first matched casevalue until a breakis encountered.

Here is a *pseudocode* example:

switch(num) {  
  case value1:  
    statement1;  
    break;  
  case value2:  
    statement2;  
    break;  
...  
  case valueN:  
    statementN;  
    break;  
}

casevalues are tested with strict equality (===). The breaktells JavaScript to stop executing statements. If the breakis omitted, the next statement will be executed.

Write a switch statement which tests valand sets answerfor the following conditions:  
1- "alpha"  
2- "beta"  
3- "gamma"  
4- "delta"

function caseInSwitch(val) {

var answer = "";

// Only change code below this line

switch (val) {

case (1):

answer = "alpha"

break;

case (2):

answer = "beta"

break;

case (3):

answer = "gamma"

break;

case (4):

answer = "delta"

break;

}

// Only change code above this line

return answer;

}

// Change this value to test

caseInSwitch(3);

In a switchstatement you may not be able to specify all possible values as casestatements. Instead, you can add the defaultstatement which will be executed if no matching casestatements are found. Think of it like the final elsestatement in an if/elsechain.

A defaultstatement should be the last case.

switch (num) {  
  case value1:  
    statement1;  
    break;  
  case value2:  
    statement2;  
    break;  
...  
  default:  
    defaultStatement;  
    break;  
}

Write a switch statement to set answerfor the following conditions:  
"a"- "apple"  
"b"- "bird"  
"c"- "cat"  
default- "stuff"

function switchOfStuff(val) {

var answer = "";

// Only change code below this line

switch (val){

case ("a"):

answer = "apple"

break;

case ("b"):

answer = "bird"

break;

case ("c"):

answer ="cat"

break;

default:

answer ="stuff";

}

// Only change code above this line

return answer;

}

// Change this value to test

switchOfStuff(1);

If the breakstatement is omitted from a switchstatement's case, the following casestatement(s) are executed until a breakis encountered. If you have multiple inputs with the same output, you can represent them in a switchstatement like this:

switch(val) {  
  case 1:  
  case 2:  
  case 3:  
    result = "1, 2, or 3";  
    break;  
  case 4:  
    result = "4 alone";  
}

Cases for 1, 2, and 3 will all produce the same result.

Write a switch statement to set answerfor the following ranges:  
1-3- "Low"  
4-6- "Mid"  
7-9- "High"

**Note**  
You will need to have a casestatement for each number in the range.

function sequentialSizes(val) {

var answer = "";

// Only change code below this line

switch (val) {

case (1):

case (2):

case (3):

answer = "Low";

break;

case (4):

case (5):

case (6):

answer = "Mid";

break;

case (7):

case (8):

case (9):

answer = "High";

break;

}

// Only change code above this line

return answer;

}

// Change this value to test

sequentialSizes(1);

If you have many options to choose from, a switchstatement can be easier to write than many chained if/else ifstatements. The following:

if (val === 1) {  
  answer = "a";  
} else if (val === 2) {  
  answer = "b";  
} else {  
  answer = "c";  
}

can be replaced with:

switch(val) {  
  case 1:  
    answer = "a";  
    break;  
  case 2:  
    answer = "b";  
    break;  
  default:  
    answer = "c";  
}

Change the chained if/else ifstatements into a switchstatement.

function chainToSwitch(val) {

var answer = "";

// Only change code below this line

switch(val) {

case ("bob"):

answer = "Marley"

break;

case (42):

answer = "The Answer"

break;

case (1):

answer = "There is no #1"

break;

case (99):

answer = "Missed me by this much!"

break;

case (7):

answer = "Ate Nine";

break;

}

// Only change code above this line

return answer;

}

// Change this value to test

chainToSwitch(7);

You may recall from [Comparison with the Equality Operator](https://learn.freecodecamp.org/waypoint-comparison-with-the-equality-operator) that all comparison operators return a boolean trueor falsevalue.

Sometimes people use an if/else statement to do a comparison, like this:

function isEqual(a,b) {  
  if (a === b) {  
    return true;  
  } else {  
    return false;  
  }  
}

But there's a better way to do this. Since ===returns trueor false, we can return the result of the comparison:

function isEqual(a,b) {  
  return a === b;  
}

Fix the function isLessto remove the if/elsestatements.

function isLess(a, b) {

// Fix this code

return a <= b;

}

// Change these values to test

isLess(10, 15);

isLess(15, 10);

When a returnstatement is reached, the execution of the current function stops and control returns to the calling location.

**Example**

function myFun() {  
  console.log("Hello");  
  return "World";  
  console.log("byebye")  
}  
myFun();

The above outputs "Hello" to the console, returns "World", but "byebye"is never output, because the function exits at the returnstatement.

Modify the function abTestso that if aor bare less than 0the function will immediately exit with a value of undefined.

**Hint**  
Remember that [undefinedis a keyword](http://www.freecodecamp.org/challenges/understanding-uninitialized-variables" \t "_blank), not a string.

// Setup

function abTest(a, b) {

// Only change code below this line

if ( a < 0)

return undefined;

if (b <0 )

return undefined;

// Only change code above this line

return Math.round(Math.pow(Math.sqrt(a) + Math.sqrt(b), 2));

}

// Change values below to test your code

abTest(2,2);

In the casino game Blackjack, a player can gain an advantage over the house by keeping track of the relative number of high and low cards remaining in the deck. This is called [Card Counting](https://en.wikipedia.org/wiki/Card_counting).

Having more high cards remaining in the deck favors the player. Each card is assigned a value according to the table below. When the count is positive, the player should bet high. When the count is zero or negative, the player should bet low.

| **Count Change** | **Cards** |
| --- | --- |
| +1 | 2, 3, 4, 5, 6 |
| 0 | 7, 8, 9 |
| -1 | 10, 'J', 'Q', 'K', 'A' |

You will write a card counting function. It will receive a cardparameter, which can be a number or a string, and increment or decrement the global countvariable according to the card's value (see table). The function will then return a string with the current count and the string Betif the count is positive, or Holdif the count is zero or negative. The current count and the player's decision (Betor Hold) should be separated by a single space.

**Example Output**  
-3 Hold  
5 Bet

**Hint**  
Do NOT reset countto 0 when value is 7, 8, or 9.  
Do NOT return an array.  
Do NOT include quotes (single or double) in the output.

Variables which are declared within a function, as well as the function parameters have *local* scope. That means, they are only visible within that function.

Here is a function myTestwith a local variable called loc.

function myTest() {  
  var loc = "foo";  
  console.log(loc);  
}  
myTest(); // logs "foo"  
console.log(loc); // loc is not defined

locis not defined outside of the function.

Declare a local variable myVarinside myLocalScope. Run the tests and then follow the instructions commented out in the editor.

**Hint**  
Refreshing the page may help if you get stuck.

You may have heard the term objectbefore.

Objects are similar to arrays, except that instead of using indexes to access and modify their data, you access the data in objects through what are called properties.

Objects are useful for storing data in a structured way, and can represent real world objects, like a cat.

Here's a sample cat object:

var cat = {  
  "name": "Whiskers",  
  "legs": 4,  
  "tails": 1,  
  "enemies": ["Water", "Dogs"]  
};

In this example, all the properties are stored as strings, such as - "name", "legs", and "tails". However, you can also use numbers as properties. You can even omit the quotes for single-word string properties, as follows:

var anotherObject = {  
  make: "Ford",  
  5: "five",  
  "model": "focus"  
};

However, if your object has any non-string properties, JavaScript will automatically typecast them as strings.

Make an object that represents a dog called myDogwhich contains the properties "name"(a string), "legs", "tails"and "friends".

You can set these object properties to whatever values you want, as long "name"is a string, "legs"and "tails"are numbers, and "friends"is an array.

var myDog = {

"name": "Barky",

"legs": 4,

"tails":2,

"friends": ["woof", "arf"]

};

There are two ways to access the properties of an object: dot notation (.) and bracket notation ([]), similar to an array.

Dot notation is what you use when you know the name of the property you're trying to access ahead of time.

Here is a sample of using dot notation (.) to read an object's property:

var myObj = {  
  prop1: "val1",  
  prop2: "val2"  
};  
var prop1val = myObj.prop1; // val1  
var prop2val = myObj.prop2; // val2

Read in the property values of testObjusing dot notation. Set the variable hatValueequal to the object's property hatand set the variable shirtValueequal to the object's property shirt.

// Setup

var testObj = {

"hat": "ballcap",

"shirt": "jersey",

"shoes": "cleats"

};

// Only change code below this line

var hatValue = testObj.hat; // Change this line

var shirtValue = testObj.shirt; // Change this line

The second way to access the properties of an object is bracket notation ([]). If the property of the object you are trying to access has a space in its name, you will need to use bracket notation.

However, you can still use bracket notation on object properties without spaces.

Here is a sample of using bracket notation to read an object's property:

var myObj = {  
  "Space Name": "Kirk",  
  "More Space": "Spock",  
  "NoSpace": "USS Enterprise"  
};  
myObj["Space Name"]; // Kirk  
myObj['More Space']; // Spock  
myObj["NoSpace"]; // USS Enterprise

Note that property names with spaces in them must be in quotes (single or double).

Read the values of the properties "an entree"and "the drink"of testObjusing bracket notation and assign them to entreeValueand drinkValuerespectively.

// Setup

var testObj = {

"an entree": "hamburger",

"my side": "veggies",

"the drink": "water"

};

// Only change code below this line

var entreeValue = testObj["an entree"]; // Change this line

var drinkValue = testObj["the drink"]; // Change this line

Another use of bracket notation on objects is to access a property which is stored as the value of a variable. This can be very useful for iterating through an object's properties or when accessing a lookup table.

Here is an example of using a variable to access a property:

var dogs = {  
  Fido: "Mutt", Hunter: "Doberman", Snoopie: "Beagle"  
};  
var myDog = "Hunter";  
var myBreed = dogs[myDog];  
console.log(myBreed); // "Doberman"

Another way you can use this concept is when the property's name is collected dynamically during the program execution, as follows:

var someObj = {  
  propName: "John"  
};  
function propPrefix(str) {  
  var s = "prop";  
  return s + str;  
}  
var someProp = propPrefix("Name"); // someProp now holds the value 'propName'  
console.log(someObj[someProp]); // "John"

Note that we do *not* use quotes around the variable name when using it to access the property because we are using the *value* of the variable, not the *name*.

Use the playerNumbervariable to look up player 16in testObjusing bracket notation. Then assign that name to the playervariable.

// Setup

var testObj = {

12: "Namath",

16: "Montana",

19: "Unitas"

};

// Only change code below this line;

var playerNumber = 16;

var player = testObj[playerNumber];

After you've created a JavaScript object, you can update its properties at any time just like you would update any other variable. You can use either dot or bracket notation to update.

For example, let's look at ourDog:

var ourDog = {  
  "name": "Camper",  
  "legs": 4,  
  "tails": 1,  
  "friends": ["everything!"]  
};

Since he's a particularly happy dog, let's change his name to "Happy Camper". Here's how we update his object's name property:

ourDog.name = "Happy Camper";or

ourDog["name"] = "Happy Camper";

Now when we evaluate ourDog.name, instead of getting "Camper", we'll get his new name, "Happy Camper".

Update the myDogobject's name property. Let's change her name from "Coder" to "Happy Coder". You can use either dot or bracket notation.

// Example

var ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

ourDog.name = "Happy Camper";

// Setup

var myDog = {

"name": "Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

// Only change code below this line.

myDog.name = "Happy Coder";

You can add new properties to existing JavaScript objects the same way you would modify them.

Here's how we would add a "bark"property to ourDog:

ourDog.bark = "bow-wow";

or

ourDog["bark"] = "bow-wow";

Now when we evaluate ourDog.bark, we'll get his bark, "bow-wow".

Add a "bark"property to myDogand set it to a dog sound, such as "woof". You may use either dot or bracket notation.

// Example

var ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

ourDog.bark = "bow-wow";

// Setup

var myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

// Only change code below this line.

myDog.bark = "woof"

We can also delete properties from objects like this:

delete ourDog.bark;

Delete the "tails"property from myDog. You may use either dot or bracket notation.

// Setup

var myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"],

"bark": "woof"

};

// Only change code below this line.

delete myDog.tails;

Objects can be thought of as a key/value storage, like a dictionary. If you have tabular data, you can use an object to "lookup" values rather than a switchstatement or an if/elsechain. This is most useful when you know that your input data is limited to a certain range.

Here is an example of a simple reverse alphabet lookup:

var alpha = {  
  1:"Z",  
  2:"Y",  
  3:"X",  
  4:"W",  
  ...  
  24:"C",  
  25:"B",  
  26:"A"  
};  
alpha[2]; // "Y"  
alpha[24]; // "C"  
  
var value = 2;  
alpha[value]; // "Y"

Convert the switch statement into an object called lookup. Use it to look up valand assign the associated string to the resultvariable.

// Setup

function phoneticLookup(val) {

var result = "";

// Only change code below this line

var lookup = {

"alpha": "Adams",

"bravo": "Boston",

"charlie": "Chicago",

"delta": "Denver",

"echo": "Easy",

"foxtrot": "Frank",

}

result = lookup[val]

// Only change code above this line

return result;

}

// Change this value to test

phoneticLookup("charlie");

Sometimes it is useful to check if the property of a given object exists or not. We can use the .hasOwnProperty(propname)method of objects to determine if that object has the given property name. .hasOwnProperty()returns trueor falseif the property is found or not.

**Example**

var myObj = {  
  top: "hat",  
  bottom: "pants"  
};  
myObj.hasOwnProperty("top"); // true  
myObj.hasOwnProperty("middle"); // false

Modify the function checkObjto test myObjfor checkProp. If the property is found, return that property's value. If not, return "Not Found".

// Setup

var myObj = {

gift: "pony",

pet: "kitten",

bed: "sleigh"

};

function checkObj(checkProp) {

// Your Code Here

if (myObj.hasOwnProperty(checkProp))

return myObj[checkProp]

else

return "Not Found"

}

// Test your code by modifying these values

checkObj("gift");

Sometimes you may want to store data in a flexible Data Structure. A JavaScript object is one way to handle flexible data. They allow for arbitrary combinations of strings, numbers, booleans, arrays, functions, and objects.

Here's an example of a complex data structure:

var ourMusic = [  
  {  
    "artist": "Daft Punk",  
    "title": "Homework",  
    "release\_year": 1997,  
    "formats": [  
      "CD",  
      "Cassette",  
      "LP"  
    ],  
    "gold": true  
  }  
];

This is an array which contains one object inside. The object has various pieces of metadata about an album. It also has a nested "formats"array. If you want to add more album records, you can do this by adding records to the top level array.

Objects hold data in a property, which has a key-value format. In the example above, "artist": "Daft Punk"is a property that has a key of "artist"and a value of "Daft Punk".

[JavaScript Object Notation](http://www.json.org/) or JSONis a related data interchange format used to store data.

{  
  "artist": "Daft Punk",  
  "title": "Homework",  
  "release\_year": 1997,  
  "formats": [  
    "CD",  
    "Cassette",  
    "LP"  
  ],  
  "gold": true  
}

**Note**  
You will need to place a comma after every object in the array, unless it is the last object in the array.

Add a new album to the myMusicarray. Add artistand titlestrings, release\_yearnumber, and a formatsarray of strings.

var myMusic =

[{

"artist": "Billy Joel",

"title": "Piano Man",

"release\_year": 1973,

"formats": [

"CD",

"8T",

"LP"

],

"gold": true

},

{"artist": "Maggie Rogers",

"title": "Light on",

"release\_year": 2019,

"formats": ["CD", "record"]}

// Add record here

];

The sub-properties of objects can be accessed by chaining together the dot or bracket notation.

Here is a nested object:

var ourStorage = {  
  "desk": {  
    "drawer": "stapler"  
  },  
  "cabinet": {  
    "top drawer": {  
      "folder1": "a file",  
      "folder2": "secrets"  
    },  
    "bottom drawer": "soda"  
  }  
};  
ourStorage.cabinet["top drawer"].folder2; // "secrets"  
ourStorage.desk.drawer; // "stapler"

Access the myStorageobject and assign the contents of the glove boxproperty to the gloveBoxContentsvariable. Use bracket notation for properties with a space in their name.

// Setup

var myStorage = {

"car": {

"inside": {

"glove box": "maps",

"passenger seat": "crumbs"

},

"outside": {

"trunk": "jack"

}

}

};

var gloveBoxContents = myStorage.car.inside["glove box"]; // Change this line

As we have seen in earlier examples, objects can contain both nested objects and nested arrays. Similar to accessing nested objects, Array bracket notation can be chained to access nested arrays.

Here is an example of how to access a nested array:

var ourPets = [  
  {  
    animalType: "cat",  
    names: [  
      "Meowzer",  
      "Fluffy",  
      "Kit-Cat"  
    ]  
  },  
  {  
    animalType: "dog",  
    names: [  
      "Spot",  
      "Bowser",  
      "Frankie"  
    ]  
  }  
];  
ourPets[0].names[1]; // "Fluffy"  
ourPets[1].names[0]; // "Spot"

Retrieve the second tree from the variable myPlantsusing object dot and array bracket notation.

// Setup

var myPlants = [

{

type: "flowers",

list: [

"rose",

"tulip",

"dandelion"

]

},

{

type: "trees",

list: [

"fir",

"pine",

"birch"

]

}

];

// Only change code below this line

var secondTree = myPlants[1].list[1]; // Change this line

Resources:

Google

W3schools

Mozilla javascript \*\*\*\*

Why ES6

Ecma script 6 – a specification of what javascript should look like

Widely used in react

ES6

Class 7

One of the biggest problems with declaring variables with the varkeyword is that you can overwrite variable declarations without an error.

var camper = 'James';  
var camper = 'David';  
console.log(camper);  
// logs 'David'

As you can see in the code above, the campervariable is originally declared as Jamesand then overridden to be David.

In a small application, you might not run into this type of problem, but when your code becomes larger, you might accidentally overwrite a variable that you did not intend to overwrite.

Because this behavior does not throw an error, searching and fixing bugs becomes more difficult.

A new keyword called letwas introduced in ES6 to solve this potential issue with the varkeyword.

If you were to replace varwith letin the variable declarations of the code above, the result would be an error.

let camper = 'James';  
let camper = 'David'; // throws an error

This error can be seen in the console of your browser.

So unlike var, when using let, a variable with the same name can only be declared once.

Note the "use strict". This enables Strict Mode, which catches common coding mistakes and "unsafe" actions. For instance:

"use strict";  
x = 3.14; // throws an error because x is not declared

Update the code so it only uses the letkeyword.

let catName;

let quote;

function catTalk() {

"use strict";

catName = "Oliver";

quote = catName + " says Meow!";

}

catTalk();

When you declare a variable with the varkeyword, it is declared globally, or locally if declared inside a function.

The letkeyword behaves similarly, but with some extra features. When you declare a variable with the letkeyword inside a block, statement, or expression, its scope is limited to that block, statement, or expression.

For example:

var numArray = [];  
for (var i = 0; i < 3; i++) {  
  numArray.push(i);  
}  
console.log(numArray);  
// returns [0, 1, 2]  
console.log(i);  
// returns 3

With the varkeyword, iis declared globally. So when i++is executed, it updates the global variable. This code is similar to the following:

var numArray = [];  
var i;  
for (i = 0; i < 3; i++) {  
  numArray.push(i);  
}  
console.log(numArray);  
// returns [0, 1, 2]  
console.log(i);  
// returns 3

This behavior will cause problems if you were to create a function and store it for later use inside a for loop that uses the ivariable. This is because the stored function will always refer to the value of the updated global ivariable.

var printNumTwo;  
for (var i = 0; i < 3; i++) {  
  if(i === 2){  
    printNumTwo = function() {  
      return i;  
    };  
  }  
}  
console.log(printNumTwo());  
// returns 3

As you can see, printNumTwo()prints 3 and not 2. This is because the value assigned to iwas updated and the printNumTwo()returns the global iand not the value ihad when the function was created in the for loop. The letkeyword does not follow this behavior:

'use strict';  
let printNumTwo;  
for (let i = 0; i < 3; i++) {  
  if (i === 2) {  
    printNumTwo = function() {  
      return i;  
    };  
  }  
}  
console.log(printNumTwo());  
// returns 2  
console.log(i);  
// returns "i is not defined"

iis not defined because it was not declared in the global scope. It is only declared within the for loop statement. printNumTwo()returned the correct value because three different ivariables with unique values (0, 1, and 2) were created by the letkeyword within the loop statement.

Fix the code so that ideclared in the if statement is a separate variable than ideclared in the first line of the function. Be certain not to use the varkeyword anywhere in your code.

This exercise is designed to illustrate the difference between how varand letkeywords assign scope to the declared variable. When programming a function similar to the one used in this exercise, it is often better to use different variable names to avoid confusion.

function checkScope() {

"use strict";

let i = "function scope";

if (true) {

let i = "block scope";

console.log("Block scope i is: ", i);

}

console.log("Function scope i is: ", i);

return i;

}

letis not the only new way to declare variables. In ES6, you can also declare variables using the constkeyword.

consthas all the awesome features that lethas, with the added bonus that variables declared using constare read-only. They are a constant value, which means that once a variable is assigned with const, it cannot be reassigned.

"use strict"  
const FAV\_PET = "Cats";  
FAV\_PET = "Dogs"; // returns error

As you can see, trying to reassign a variable declared with constwill throw an error. You should always name variables you don't want to reassign using the constkeyword. This helps when you accidentally attempt to reassign a variable that is meant to stay constant. A common practice when naming constants is to use all uppercase letters, with words separated by an underscore.

Change the code so that all variables are declared using letor const. Use letwhen you want the variable to change, and constwhen you want the variable to remain constant. Also, rename variables declared with constto conform to common practices, meaning constants should be in all caps.

function printManyTimes(str) {

"use strict";

// change code below this line

"use strict"

const SENTENCE = str + " is cool!";

for(let i = 0; i < str.length; i+=2) {

console.log(SENTENCE);

}

// change code above this line

}

printManyTimes("freeCodeCamp");

The constdeclaration has many use cases in modern JavaScript.

Some developers prefer to assign all their variables using constby default, unless they know they will need to reassign the value. Only in that case, they use let.

However, it is important to understand that objects (including arrays and functions) assigned to a variable using constare still mutable. Using the constdeclaration only prevents reassignment of the variable identifier.

"use strict";  
const s = [5, 6, 7];  
s = [1, 2, 3]; // throws error, trying to assign a const  
s[2] = 45; // works just as it would with an array declared with var or let  
console.log(s); // returns [5, 6, 45]

As you can see, you can mutate the object [5, 6, 7]itself and the variable swill still point to the altered array [5, 6, 45]. Like all arrays, the array elements in sare mutable, but because constwas used, you cannot use the variable identifier sto point to a different array using the assignment operator.

An array is declared as const s = [5, 7, 2]. Change the array to [2, 5, 7]using various element assignment.

const s = [5, 7, 2];

function editInPlace() {

"use strict";

// change code below this line

s[0] = 2;

s[1] = 5;

s[2] = 7;

// s = [2, 5, 7]; <- this is invalid

// change code above this line

}

editInPlace();

As seen in the previous challenge, constdeclaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function Object.freezeto prevent data mutation.

Once the object is frozen, you can no longer add, update, or delete properties from it. Any attempt at changing the object will be rejected without an error.

let obj = {  
  name:"FreeCodeCamp",  
  review:"Awesome"  
};  
Object.freeze(obj);  
obj.review = "bad"; //will be ignored. Mutation not allowed  
obj.newProp = "Test"; // will be ignored. Mutation not allowed  
console.log(obj);  
// { name: "FreeCodeCamp", review:"Awesome"}

In this challenge you are going to use Object.freezeto prevent mathematical constants from changing. You need to freeze the MATH\_CONSTANTSobject so that no one is able alter the value of PI, add, or delete properties .

function freezeObj() {

"use strict";

const MATH\_CONSTANTS = {

PI: 3.14

};

// change code below this line

Object.freeze(MATH\_CONSTANTS);

// change code above this line

try {

MATH\_CONSTANTS.PI = 99;

} catch( ex ) {

console.log(ex);

}

return MATH\_CONSTANTS.PI;

}

const PI = freezeObj();

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we do not reuse them anywhere else.

To achieve this, we often use the following syntax:

const myFunc = function() {  
  const myVar = "value";  
  return myVar;  
}

ES6 provides us with the syntactic sugar to not have to write anonymous functions this way. Instead, you can use **arrow function syntax**:

const myFunc = () => {  
  const myVar = "value";  
  return myVar;  
}

When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword returnas well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:

const myFunc = () => "value"

This code will still return valueby default.

Rewrite the function assigned to the variable magicwhich returns a new Date()to use arrow function syntax. Also make sure nothing is defined using the keyword var.

const magic = () => new Date ()

Just like a normal function, you can pass arguments into arrow functions.

// doubles input value and returns it  
const doubler = (item) => item \* 2;

You can pass more than one argument into arrow functions as well.

Rewrite the myConcatfunction which appends contents of arr2to arr1so that the function uses arrow function syntax.

const arr1 = [1,2]

const arr2 = [3,4,5]

const myConcat = (arr1) => arr1 + arr2;

// test your code

console.log(myConcat([1, 2], [3, 4, 5]));

It's time we see how powerful arrow functions are when processing data.

Arrow functions work really well with higher order functions, such as map(), filter(), and reduce(), that take other functions as arguments for processing collections of data.

Read the following code:

FBPosts.filter(function(post) {  
  return post.thumbnail !== null && post.shares > 100 && post.likes > 500;  
})

We have written this with filter()to at least make it somewhat readable. Now compare it to the following code which uses arrow function syntax instead:

FBPosts.filter((post) => post.thumbnail !== null && post.shares > 100 && post.likes > 500)

This code is more succinct and accomplishes the same task with fewer lines of code.

Use arrow function syntax to compute the square of only the positive integers (decimal numbers are not integers) in the array realNumberArrayand store the new array in the variable squaredIntegers.

Source = “WebCo-Class-1/photo\_name