Comment your JavaScript Code

Comments are lines of code that JavaScript will intentionally ignore. Comments are a great way to leave notes to yourself and to other people who will later need to figure out what that code does.

There are two ways to write comments in JavaScript:

Using // will tell JavaScript to ignore the remainder of the text on the current line:

// This is an in-line comment.

You can make a multi-line comment beginning with /\*and ending with \*/:

/\* This is a   
 multi-line comment \*/

**Best Practice**  
As you write code, you should regularly add comments to clarify the function of parts of your code. Good commenting can help communicate the intent of your code—both for others *and* for your future self.

Instructions

Try creating one of each type of comment.

Top of Form

Bottom of Form

Create a // style comment that contains at least five letters.

Create a /\* \*/ style comment that contains at least five letters.

Make sure that you close the comment with a \*/.

Code : //sameer\_coder...

/\*sameer is a back end developer..

...\*/

Declare JavaScript Variables

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.

Variables are 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 variables differ 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 var in front of it, like so:

var ourName;

creates a variable called ourName. In JavaScript we end statements with semicolons.

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

Instructions

Use the var keyword to create a variable called myName.

**Hint**  
Look at the ourName example if you get stuck.

You should declare myName with the var keyword, ending with a semicolon

Code:

// Example

var ourName;

// Define myName below this line

var myName;

Storing Values with the Assignment Operator

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

myVariable = 5;

Assigns the Number value 5 to 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;

Assigns 5 to myVar and then resolves myVar to 5again and assigns it to myNum.

Instructions

Assign the value 7 to variable a.

Assign the contents of a to variable b.

Do not change code above the line

a should have a value of 7

b should have a value of 7

a should be assigned to b with

Code:

// Setup

var a;

var b = 2;

// Only change code below this line

var a = 7;

b = a;

Initializing Variables with the Assignment Operator

It is common to *initialize* a variable to an initial value in the same line as it is declared.

var myVar = 0;

Creates a new variable called myVar and assigns it an initial value of 0.

Instructions

Define a variable a with var and initialize it to a value of 9.

Code: // Example

var ourVar = 19;

// Only change code below this line

var a = 9;

Understanding Uninitialized Variables

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

Instructions

Initialize the three variables a, b, and c with 5, 10, and "I am a" respectively so that they will not be undefined.

Code:

// Initialize these three variables

var a = 5;

var b = 10;

var c = "I am a";

// Do not change code below this line

a = a + 1;

b = b + 5;

c = c + " String!";

Understanding Case Sensitivity in Variables

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

MYVAR is not the same as MyVar nor 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.

**Examples:**

var someVariable;  
var anotherVariableName;  
var thisVariableNameIsTooLong;

Instructions

Modify the existing declarations and assignments so their names use *camelCase*.  
Do not create any new variables.

studlyCapVar is defined and has a value of 10

properCamelCase is defined and has a value of "A String"

titleCaseOver is defined and has a value of 9000

studlyCapVar should use camelCase in both declaration and assignment sections.

properCamelCase should use camelCase in both declaration and assignment sections.

titleCaseOver should use camelCase in both declaration and assignment sections.

Code:

// Declarations

var studlyCapVar = 10;

var properCamelCase = "A String";

var titleCaseOver = 9000;

// Assignments

studlyCapVar = 10;

properCamelCase = "A String";

titleCaseOver = 9000;

Add Two Numbers with JavaScript

Number is a data type in JavaScript which represents numeric data.

Now let's try to add two numbers using JavaScript.

JavaScript uses the + symbol as addition operation when placed between two numbers.

**Example**

myVar = 5 + 10; // assigned 15

Instructions

Change the 0 so that sum will equal 20.

Code:

var sum = 10 + 10;

Subtract One Number from Another with JavaScript

We can also subtract one number from another.

JavaScript uses the - symbol for subtraction.

**Example**

myVar = 12 - 6; // assigned 6

Instructions

Change the 0 so the difference is 12.

Code:

var difference = 45 - 33;

Multiply Two Numbers with JavaScript

We can also multiply one number by another.

JavaScript uses the \* symbol for multiplication of two numbers.

**Example**

myVar = 13 \* 13; // assigned 169

Instructions

Change the 0 so that product will equal 80.

Code:

var product = 8 \* 10;

Divide One Number by Another with JavaScript

We can also divide one number by another.

JavaScript uses the / symbol for division.

**Example**

myVar = 16 / 2; // assigned 8

Instructions

Change the 0 so that the quotient is equal to 2.

Code:

var quotient = 66 / 33;

Increment a Number with JavaScript

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.

Instructions

Change the code to use the ++ operator on myVar.

**Hint**  
Learn more about [**Arithmetic operators - Increment (++)**](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment_()).

Code:

var myVar = 87;

// Only change code below this line

myVar++;

Decrement a Number with JavaScript

You can easily *decrement* or decrease a variable by one with the -- operator.

i--;

is the equivalent of

i = i - 1;

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

Instructions

Change the code to use the -- operator on myVar.

Code:

var myVar = 11;

// Only change code below this line

myVar--;

Create Decimal Numbers with JavaScript

We can store decimal numbers in variables too. Decimal numbers are sometimes referred to as *floating point*numbers or *floats*.

**Note**  
Not all real numbers can accurately be represented in *floating point*. This can lead to rounding errors. [**Details Here**](https://en.wikipedia.org/wiki/Floating_point#Accuracy_problems).

Instructions

Create a variable myDecimal and give it a decimal value with a fractional part (e.g. 5.7).

Code:

var ourDecimal = 5.7;

// Only change code below this line

var myDecimal = 3.0987;

Multiply Two Decimals with JavaScript

In JavaScript, you can also perform calculations with decimal numbers, just like whole numbers.

Let's multiply two decimals together to get their product.

Instructions

Change the 0.0 so that product will equal 5.0.

Code:

var product = 2.0 \* 2.5;

Divide one Decimal by Another with JavaScript

Now let's divide one decimal by another.

Instructions

Change the 0.0 so that quotient will equal to 2.2.

Code:

var quotient = 4.4 / 2.0;

Finding a Remainder in JavaScript

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 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.

Instructions

Set remainder equal to the remainder of 11 divided by 3 using the *remainder* (%) operator.

Code: var remainder = 11 % 3;

Compound Assignment With Augmented Addition

In programming, it is common to use assignments to modify the contents of a variable. Remember that everything to the right of the equals sign is evaluated first, so we can say:

myVar = myVar + 5;

to add 5 to 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.

myVar += 5; will add 5 to myVar.

Instructions

Convert the assignments for a, b, and c to use the +=operator.

Code:

var a = 3;

var b = 17;

var c = 12;

// Only modify code below this line

a += 12;

b += 9;

c += 7;

Compound Assignment With Augmented Subtraction

Like the += operator, -= subtracts a number from a variable.

myVar = myVar - 5;

will subtract 5 from myVar. This can be rewritten as:

myVar -= 5;

Instructions

Convert the assignments for a, b, and c to use the -=operator.

Code:

var a = 11;

var b = 9;

var c = 3;

// Only modify code below this line

a -= 6;

b -= 15;

c -= 1;

Compound Assignment With Augmented Multiplication

The \*= operator multiplies a variable by a number.

myVar = myVar \* 5;

will multiply myVar by 5. This can be rewritten as:

myVar \*= 5;

Instructions

Convert the assignments for a, b, and c to use the \*=operator.

Code:

var a = 5;

var b = 12;

var c = 4.6;

// Only modify code below this line

a \*= 5;

b \*= 3;

c \*= 10;

Compound Assignment With Augmented Division

The /= operator divides a variable by another number.

myVar = myVar / 5;

Will divide myVar by 5. This can be rewritten as:

myVar /= 5;

Instructions

Convert the assignments for a, b, and c to use the /=operator.

Code:

var a = 48;

var b = 108;

var c = 33;

// Only modify code below this line

a /= 12;

b /= 4;

c /= 11;

Convert Celsius to Fahrenheit

To test your learning, you will create a solution "from scratch". Place your code between the indicated lines and it will be tested against multiple test cases.

The algorithm to convert from Celsius to Fahrenheit is the temperature in Celsius times 9/5, plus 32.

You are given a variable celsius representing a temperature in Celsius. Use the variable fahrenheitalready defined and apply the algorithm to assign it the corresponding temperature in Fahrenheit.

**Note**  
Don't worry too much about the function and returnstatements as they will be covered in future challenges. For now, only use operators that you have already learned.

convertToF(0) should return a number

convertToF(-30) should return a value of -22

convertToF(-10) should return a value of 14

convertToF(0) should return a value of 32

convertToF(20) should return a value of 68

convertToF(30) should return a value of 86

code:

function convertToF(celsius) {

var fahrenheit;

// Only change code below this line

fahrenheit = celsius \* (9/5) + 32;

// Only change code above this line

return fahrenheit;

}

// Change the inputs below to test your code

convertToF(0);

Declare String Variables

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.

Instructions

Create two new string variables: myFirstName and myLastName and assign them the values of your first and last name, respectively.

Code: var myFirstName = "sameer";

var myLastName = "loophole";

Escaping Literal Quotes in Strings

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".

Instructions

Use *backslashes* to assign a string to the myStrvariable so that *if* you were to print it to the console, you would see:

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

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

Quoting Strings with Single Quotes

*String* values in JavaScript may be written with single or double quotes, so long as you start and end with the same type of quote. Unlike some languages, single and double quotes are functionally identical in JavaScript.

"This string has \"double quotes\" in it"

The value in using one or the other has to do with the need to *escape* quotes of the same type. Unless they are escaped, you cannot have more than one pair of whichever quote type begins a string.

If you have a string with many double quotes, this can be difficult to read and write. Instead, use single quotes:

'This string has "double quotes" in it. And "probably" lots of them.'

Instructions

Change the provided string from double to single quotes and remove the escaping.

Remove all the backslashes (\)

You should have two single quotes ' and four double quotes "

Only remove the backslashes \ used to escape quotes.

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

Escape Sequences in Strings

Quotes are not the only characters that can be *escaped*inside a string. Here is a table of common escape sequences:

| **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.*

Instructions

Assign the following three lines of text into the single variable myStr using 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.

FirstLinenewlinebackslashSecondLinebackslashcarriage-returnThirdLine

Top of Form

Bottom of Form

Bottom of Form

myStr should have encoded text with the proper escape sequences and no spacing.

myStr should have one newline character \n

myStr should have one carriage return character \r

myStr should have two correctly escaped backslash characters \\

code:

var myStr = "FirstLine\n\\SecondLine\\\rThirdLine"; // Change this line

Concatenating Strings with Plus Operator

In JavaScript, when the + operator is used with a String value, 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.

Instructions

Build myStr from the strings "This is the start. "and "This is the end." using the + operator.

Code:

// Example

var ourStr = "I come first. " + "I come second.";

// Only change code below this line

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

Concatenating Strings with the Plus Equals Operator

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.

Instructions

Build myStr over several lines by concatenating these two strings:  
"This is the first sentence. " and "This is the second sentence." using the += operator.

Code:

// Example

var ourStr = "I come first. ";

ourStr += "I come second.";

// Only change code below this line

var myStr = "This is the first sentence. ";

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

Constructing Strings with Variables

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.

Instructions

Set myName to a string equal to your name and build myStr with myName between the strings "My name is " and " and I am swell!"

Code:

// Example

var ourName = "Free Code Camp";

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

// Only change code below this line

var myName = "Sameer";

var myStr = "My name is " + myName + "and I am swell!";

Appending Variables to Strings

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.

Instructions

Set someAdjective and append it to myStr using the += operator.

Code:

// Example

var anAdjective = "awesome!";

var ourStr = "Free Code Camp is ";

ourStr += anAdjective;

// Only change code below this line

var someAdjective = "like falling in a Loophole.";

var myStr = "Learning to code is ";

myStr += someAdjective;

Find the Length of a String

You can find the length of a String value by writing .length after 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.length property.

Instructions

Use the .length property to count the number of characters in the lastName variable and assign it to lastNameLength.

Code:

// Example

var firstNameLength = 0;

var firstName = "Ada";

firstNameLength = firstName.length;

// Setup

var lastNameLength = 0;

var lastName = "Lovelace";

// Only change code below this line.

lastNameLength = lastName.length;

Use Bracket Notation to Find the First Character in a String

Bracket notation is a way to get a character at a specific index within 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].

Instructions

Use *bracket notation* to find the first character in the lastName variable and assign it to firstLetterOfLastName.

**Hint**  
Try looking at the firstLetterOfFirstName variable declaration if you get stuck.

Code:

// Example

var firstLetterOfFirstName = "";

var firstName = "Ada";

firstLetterOfFirstName = firstName[0];

// Setup

var firstLetterOfLastName = "";

var lastName = "Lovelace";

// Only change code below this line

firstLetterOfLastName = lastName[0];

Understand String Immutability

In JavaScript, String values 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 myStr to "Job", because the contents of myStr cannot be altered. Note that this does *not* mean that myStr cannot be changed, just that the individual characters of a *string literal* cannot be changed. The only way to change myStr would be to assign it with a new string, like this:

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

Instructions

Correct the assignment to myStr to achieve the desired effect.

Code:

// Setup

var myStr = "Jello World";

// Only change code below this line

myStr = "Hello World";

Use Bracket Notation to Find the Nth Character in a String

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.

Instructions

Let's try to set thirdLetterOfLastName to equal the third letter of the lastName variable using bracket notation.

**Hint**  
Try looking at the secondLetterOfFirstName variable declaration if you get stuck.

Code:

// Example

var firstName = "Ada";

var secondLetterOfFirstName = firstName[1];

// Setup

var lastName = "Lovelace";

// Only change code below this line.

var thirdLetterOfLastName = lastName[2];

Use Bracket Notation to Find the Last Character in a String

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].

Instructions

Use *bracket notation* to find the last character in the lastName variable.

**Hint**  
Try looking at the lastLetterOfFirstName variable declaration if you get stuck.

Code:

// Example

var firstName = "Ada";

var lastLetterOfFirstName = firstName[firstName.length - 1];

// Setup

var lastName = "Lovelace";

// Only change code below this line.

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

Use Bracket Notation to Find the NthtoLast Character in a String

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]

Instructions

Use *bracket notation* to find the second-to-last character in the lastName string.

**Hint**  
Try looking at the thirdToLastLetterOfFirstNamevariable declaration if you get stuck.

Code:

// Example

var firstName = "Ada";

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

// Setup

var lastName = "Lovelace";

// Only change code below this line

var secondToLastLetterOfLastName = lastName[lastName.length - 2];

Word Blanks

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.

You will need to use string operators to build a new string, result, using the provided variables: myNoun, myAdjective, myVerb, and myAdverb.

You will also need to use additional strings, which will not change, and must be in between all of the provided words. The output should be a complete sentence.

We have provided a framework for testing your results with different words. The tests will run your function with several different inputs to make sure all of the provided words appear in the output, as well as your extra strings.

wordBlanks("","","","") should return a string.

wordBlanks("dog", "big", "ran", "quickly") should contain all of the passed in words separated by non-word characters (and any additional words in your madlib).

wordBlanks("cat", "little", "hit", "slowly") should contain all of the passed in words separated by non-word characters (and any additional words in your madlib).

Top of Form

Bottom of Form

Code:

function wordBlanks(myNoun, myAdjective, myVerb, myAdverb) {

var result = "";

// Your code below this line

result = "A " + myNoun + " which is " + myAdjective +" njdfhvf " + myVerb + " djehfuv "+ myAdverb;

// Your code above this line

return result;

}

// Change the words here to test your function

wordBlanks("lion", "little", "hit", "slowly");

Store Multiple Values in one Variable using JavaScript Arrays

With JavaScript array variables, 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"].

Instructions

Modify the new array myArray so that it contains both a string and a number (in that order).

**Hint**  
Refer to the example code in the text editor if you get stuck.

Code:

// Example

var ourArray = ["John", 23];

// Only change code below this line.

var myArray = ["Loophole",9];

Nest one Array within Another Array

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

Instructions

Create a nested array called myArray.

Code:

// Example

var ourArray = [["the universe", 42], ["everything", 101010]];

// Only change code below this line.

var myArray = [["I'm gonna go work in google","hahaha"],["That's for sure","cool"]];

Access Array Data with Indexes

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 = [1,2,3];  
array[0]; // equals 1  
var data = array[1]; // equals 2

Instructions

Create a variable called myData and set it to equal the first value of myArray using bracket notation.

Code:

// Example

var ourArray = [1,2,3];

var ourData = ourArray[0]; // equals 1

// Setup

var myArray = [1,2,3];

var myData = myArray[0];

// Only change code below this line.

Modify Array Data With Indexes

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

**Example**

var ourArray = [3,2,1];  
ourArray[0] = 1; // equals [1,2,1]

Instructions

Modify the data stored at index 0 of myArray to a value of 3.

Code:

// Example

var ourArray = [1,2,3];

ourArray[1] = 3; // ourArray now equals [1,3,3].

// Setup

var myArray = [1,2,3];

myArray[0] = 3;

// Only change code below this line.

Access MultiDimensional Arrays With Indexes

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

Instructions

Using bracket notation select an element from myArraysuch that myData is equal to 8.

Code:

// Setup

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

// Only change code below this line.

var myData = myArray[2][1];

Manipulate Arrays With push

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]

Instructions

Push ["dog", 3] onto the end of the myArray variable

Code:

// Example

var ourArray = ["Stimpson", "J", "cat"];

ourArray.push(["happy", "joy"]);

// ourArray now equals ["Stimpson", "J", "cat", ["happy", "joy"]]

// Setup

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

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

// Only change code below this line.

Manipulate Arrays With pop

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.

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

For example, for the code  
var oneDown = [1, 4, 6].pop();  
the variable oneDown now holds the value 6 and the array becomes [1, 4].

Instructions

Use the .pop() function to remove the last item from myArray, assigning the "popped off" value to removedFromMyArray.

Code:

// Example

var ourArray = [1,2,3];

var removedFromOurArray = ourArray.pop();

// removedFromOurArray now equals 3, and ourArray now equals [1,2]

// Setup

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

// Only change code below this line.

var removedFromMyArray = myArray.pop();

Manipulate Arrays With shift

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.

Instructions

Use the .shift() function to remove the first item from myArray, assigning the "shifted off" value to removedFromMyArray.

Code:

// Example

var ourArray = ["Stimpson", "J", ["cat"]];

removedFromOurArray = ourArray.shift();

// removedFromOurArray now equals "Stimpson" and ourArray now equals ["J", ["cat"]].

// Setup

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

// Only change code below this line.

var removedFromMyArray = myArray.shift();

Manipulate Arrays With unshift

Not only can you shift elements off of the beginning of an array, you can also unshift elements 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.

Instructions

Add ["Paul",35] to the beginning of the myArrayvariable using unshift().

Code:

// Example

var ourArray = ["Stimpson", "J", "cat"];

ourArray.shift(); // ourArray now equals ["J", "cat"]

ourArray.unshift("Happy");

// ourArray now equals ["Happy", "J", "cat"]

// Setup

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

myArray.shift();

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

// Only change code below this line.

Shopping List

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 lis

Code:

var myList = [];

myList.push(["Cockroaches",32],["Frogs",9],["GrassHoppers",12],["Chameleons",22],["Spiders",99]);

Write Reusable JavaScript with Functions

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.

Instructions

Create a function called reusableFunction which prints "Hi World" to the dev console.

Call the function.

Code:

// Example

function ourReusableFunction() {

console.log("Heyya, World");

}

ourReusableFunction();

// Only change code below this line

function reusableFunction() {

console.log("Hi World");

}

reusableFunction();

Passing Values to Functions with Arguments

*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, param1 and 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, param1 will equal "Hello" and param2 will equal "World". Note that you could call testFun again with different arguments and the parameters would take on the value of the new arguments.

Instructions

Create a function called functionWithArgs that accepts two arguments and outputs their sum to the dev console.

Call the function.

Code:

// Example

function ourFunctionWithArgs(a, b) {

console.log(a - b);

}

ourFunctionWithArgs(10, 5); // Outputs 5

// Only change code below this line.

function functionWithArgs(x,y) {

console.log(x+y);

}

functionWithArgs(3,4);

Global Scope and Functions

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 var keyword are automatically created in the global scope. This can create unintended consequences elsewhere in your code or when running a function again. You should always declare your variables with var.

Instructions

Using var, declare a global variable myGlobal outside of any function. Initialize it with a value of 10.

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

myGlobal should be defined

myGlobal should have a value of 10

myGlobal should be declared using the var keyword

oopsGlobal should have a value of 5

Do not declare oopsGlobal using the var keyword

Code: var myGlobal = 10;

// Declare your variable here

function fun1() {

// Assign 5 to oopsGlobal Here

oopsGlobal = 5;

}

// 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);

}

Local Scope and Functions

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 myTest with a local variable called loc.

function myTest() {  
 var loc = "foo";  
 console.log(loc);  
}  
myTest(); // "foo"  
console.log(loc); // "undefined"

loc is not defined outside of the function.

Instructions

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

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

Code:

function myLocalScope() {

'use strict';

var myVar;

console.log(myVar);

}

myLocalScope();

// Run and check the console

// myVar is not defined outside of myLocalScope

//console.log(myVar);

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

Global vs Local Scope in Functions

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

In this example:

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

The function myFun will return "Head" because the local version of the variable is present.

Instructions

Add a local variable to myOutfit to override the value of outerWear with "sweater".

Code:

// Setup

var outerWear = "T-Shirt";

function myOutfit() {

// Only change code below this line

var outerWear = "sweater";

// Only change code above this line

return outerWear;

}

myOutfit();

Return a Value from a Function with Return

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

**Example**

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

plusThree takes an *argument* for num and returns a value equal to num + 3.

Instructions

Create a function timesFive that accepts one argument, multiplies it by 5, and returns the new value.

Code:

// Example

function minusSeven(num) {

return num - 7;

}

// Only change code below this line

function timesFive(num1) {

return num1 \* 5; }

Assignment with a Returned Value

If you'll recall from our discussion of [**Storing Values with the Assignment Operator**](https://www.freecodecamp.com/challenges/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 sum which adds two numbers together, then:

ourSum = sum(5, 12);

will call sum function, which returns a value of 17 and assigns it to ourSum variable.

Instructions

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

Code:

// 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);

Stand in Line

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 queue and old items are taken off from the front of the queue.

Write a function nextInLine which 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 array. The nextInLine function should then return the element that was removed.

nextInLine([], 1) should return 1

nextInLine([2], 1) should return 2

nextInLine([5,6,7,8,9], 1) should return 5

After nextInLine(testArr, 10), testArr[4] should be 10

Code:

function nextInLine(arr, item) {

// Your code here

arr.push(item);

return arr.shift(); // Change this line

}

// Test Setup

var testArr = [1,2,3,4,5];

// Display Code

console.log("Before: " + JSON.stringify(testArr));

console.log(nextInLine(testArr, 6)); // Modify this line to test

console.log("After: " + JSON.stringify(testArr));

Understanding Boolean Values

Another data type is the *Boolean*. Booleans may only be one of two values: true or false. They are basically little on-off switches, where true is "on" and false is "off." These two states are mutually exclusive.

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

Instructions

Modify the welcomeToBooleans function so that it returns true instead of false when the run button is clicked.

Code:

function welcomeToBooleans() {

// Only change code below this line.

return true; // Change this line

// Only change code above this line.

}

Use Conditional Logic with If Statements

If statements are used to make decisions in code. The keyword if tells JavaScript to execute the code in the curly braces under certain conditions, defined in the parentheses. These conditions are known as Booleanconditions because they may only be true or 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 test is called with a value of true, the ifstatement evaluates myCondition to see if it is true or not. Since it is true, the function returns "It was true". When we call test with a value of false, myCondition is *not* true and the statement in the curly braces is not executed and the function returns "It was false".

Instructions

Create an if statement inside the function to return "Yes, that was true" if the parameter wasThatTrue is true and return "No, that was false" otherwise.

Code:

// Example

function ourTrueOrFalse(isItTrue) {

if (isItTrue) {

return "Yes, it's true";

}

return "No, it's false";

}

// 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(false);

Comparison with the Equality Operator

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

The most basic operator is the equality operator ==. The equality operator compares two values and returns trueif they're equivalent or false if 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 myVal is 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, numbers and strings), it must convert one type to another. Once it does, however, it can compare terms as follows:

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

Instructions

Add the equality operator to the indicated line so that the function will return "Equal" when val is equivalent to 12

Code:

// Setup

function testEqual(val) {

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

return "Equal";

}

return "Not Equal";

}

// Change this value to test

testEqual("12");

Comparison with the Strict Equality Operator

Strict equality (===) is the counterpart to the equality operator (==). Unlike the equality operator, strict equality tests both the data type and value of the compared elements.

**Examples**

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

In the second example, 3 is a Number type and '3' is a String type.

Instructions

Use the strict equality operator in the if statement so the function will return "Equal" when val is strictly equal to 7

Code:

// Setup

function testStrict(val) {

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

return "Equal";

}

return "Not Equal";

}

// Change this value to test

testStrict("7");

Comparison with the Inequality Operator

The inequality operator (!=) is the opposite of the equality operator. It means "Not Equal" and returns false where equality would return true and *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

Instructions

Add the inequality operator != in the if statement so that the function will return "Not Equal" when val is not equivalent to 99

testNotEqual(99) should return "Equal"

testNotEqual("99") should return "Equal"

testNotEqual(12) should return "Not Equal"

testNotEqual("12") should return "Not Equal"

testNotEqual("bob") should return "Not Equal"

You should use the != operator

Code:

// Setup

function testNotEqual(val) {

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

return "Not Equal";

}

return "Equal";

}

// Change this value to test

testNotEqual(10);

Comparison with the Strict Inequality Operator

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

**Examples**

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

Instructions

Add the strict inequality operator to the ifstatement so the function will return "Not Equal" when val is not strictly equal to 17

testStrictNotEqual(17) should return "Equal"

testStrictNotEqual("17") should return "Not Equal"

testStrictNotEqual(12) should return "Not Equal"

testStrictNotEqual("bob") should return "Not Equal"

You should use the !== operator

code:

// 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(10);

Comparison with the Greater Than Operator

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

Instructions

Add the greater than operator to the indicated lines so that the return statements make sense.  
testGreaterThan(0) should return "10 or Under"

testGreaterThan(10) should return "10 or Under"

testGreaterThan(11) should return "Over 10"

testGreaterThan(99) should return "Over 10"

testGreaterThan(100) should return "Over 10"

testGreaterThan(101) should return "Over 100"

testGreaterThan(150) should return "Over 100"

You should use the > operator at least twice

Code:

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(10);

Comparison with the Greater Than Or Equal To Operator

The greater than or equal to operator (>=) 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

Instructions

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

testGreaterOrEqual(0) should return "9 or Under"

testGreaterOrEqual(9) should return "9 or Under"

testGreaterOrEqual(10) should return "10 or Over"

testGreaterOrEqual(11) should return "10 or Over"

testGreaterOrEqual(19) should return "10 or Over"

testGreaterOrEqual(100) should return "20 or Over"

testGreaterOrEqual(21) should return "20 or Over"

You should use the >= operator at least twice

Code:

function testGreaterOrEqual(val) {

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

return "20 or Over";

}

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

return "10 or Over";

}

return "9 or Under";

}

// Change this value to test

testGreaterOrEqual(20);

Comparison with the Less Than Operator

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

Instructions

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

testLessThan(0) should return "Under 25"

testLessThan(24) should return "Under 25"

testLessThan(25) should return "Under 55"

testLessThan(54) should return "Under 55"

testLessThan(55) should return "55 or Over"

testLessThan(99) should return "55 or Over"

You should use the < operator at least twice

Code:

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);

Comparison with the Less Than Or Equal To Operator

The less than or equal to operator (<=) compares the values of two numbers. If the number to the left is less than or equal the number to the right, it returns true. If the number on the left is greater than the number on the right, it returns false. Like the equality operator, less than or equal to converts data types.

**Examples**

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

Instructions

Add the less than or equal to operator to the indicated lines so that the return statements make s

testLessOrEqual(0) should return "Smaller Than or Equal to 12"

testLessOrEqual(11) should return "Smaller Than or Equal to 12"

testLessOrEqual(12) should return "Smaller Than or Equal to 12"

testLessOrEqual(23) should return "Smaller Than or Equal to 24"

testLessOrEqual(24) should return "Smaller Than or Equal to 24"

testLessOrEqual(25) should return "25 or More"

testLessOrEqual(55) should return "25 or More"

You should use the <= operator at least twice

Code:

function testLessOrEqual(val) {

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

return "Smaller Than or Equal to 12";

}

if (val <= 24) { // Change this line

return "Smaller Than or Equal to 24";

}

return "25 or More";

}

// Change this value to test

testLessOrEqual(10);

Comparisons with the Logical And Operator

Sometimes you will need to test more than one thing at a time. The *logical and* operator (&&) returns true if 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 num is between 6 and 9 (6 and 9 included). The same logic can be written as:

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

Instructions

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

You should use the && operator once

You should only have one if statement

testLogicalAnd(0) should return "No"

testLogicalAnd(24) should return "No"

testLogicalAnd(25) should return "Yes"

testLogicalAnd(30) should return "Yes"

testLogicalAnd(50) should return "Yes"

testLogicalAnd(51) should return "No"

testLogicalAnd(75) should return "No"

testLogicalAnd(80) should return "No"

code:

function logicalOperator(val) {

if (val >= 21 && val <= 30) {

return "next time";

}

return "just clear the skies";

}

console.log(logicalOperator(10));Top of Form

Comparisons with the Logical Or Operator

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

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 num is between 5 and 10 (5 and 10 included). The same logic can be written as:

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

Instructions

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

You should use the || operator once

You should only have one if statement

testLogicalOr(0) should return "Outside"

testLogicalOr(9) should return "Outside"

testLogicalOr(10) should return "Inside"

testLogicalOr(15) should return "Inside"

testLogicalOr(19) should return "Inside"

testLogicalOr(20) should return "Inside"

testLogicalOr(21) should return "Outside"

testLogicalOr(25) should return "Outside"

code:

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);

Introducing Else Statements

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

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

Instructions

Combine the if statements into a single if/elsestatement.

You should only have one if statement in the editor

You should use an else statement

testElse(4) should return "5 or Smaller"

testElse(5) should return "5 or Smaller"

testElse(6) should return "Bigger than 5"

testElse(10) should return "Bigger than 5"

Do not change the code above or below the lines.

Code:

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);

Introducing Else If Statements

If you have multiple conditions that need to be addressed, you can chain if statements 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";  
}

Instructions

Convert the logic to use else if statements.

Top of Form

5 and 10

Bottom of Form

You should have at least two elsestatements

You should have at least two ifstatements

testElseIf(0) should return "Smaller than 5"

testElseIf(5) should return "Between 5 and 10"

testElseIf(7) should return "Between 5 and 10"

testElseIf(10) should return "Between 5 and 10"

testElseIf(12) should return "Greater than 10"

code:

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(7);

Logical Order in If Else Statements

Order is important in if, else if statements.

The loop 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"

Instructions

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

orderMyLogic(4) should return "Less than 5"

orderMyLogic(6) should return "Less than 10"

orderMyLogic(11) should return "Greater than or equal to 10"

code:

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(6);

Chaining If Else Statements

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

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

Instructions

Write chained if/else if statements 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"

You should have at least four elsestatements

You should have at least four ifstatements

You should have at least one returnstatement

testSize(0) should return "Tiny"

testSize(4) should return "Tiny"

testSize(5) should return "Small"

testSize(8) should return "Small"

testSize(10) should return "Medium"

testSize(14) should return "Medium"

testSize(15) should return "Large"

testSize(17) should return "Large"

testSize(20) should return "Huge"

testSize(25) should return "Huge"

code:

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 {

return "Huge";

}

return "Change Me";

// Only change code above this line

}

// Change this value to test

testSize(7);

Golf Code

In the game of [**golf**](https://en.wikipedia.org/wiki/Golf) each hole has a par meaning the average number of strokes a 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 par your strokes are, there is a different nickname.

Your function will be passed par and 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!" |

par and strokes will always be numeric and positive.

golfScore(4, 1) should return "Hole-in-one!"

golfScore(4, 2) should return "Eagle"

golfScore(5, 2) should return "Eagle"

golfScore(4, 3) should return "Birdie"

golfScore(4, 4) should return "Par"

golfScore(1, 1) should return "Hole-in-one!"

golfScore(5, 5) should return "Par"

golfScore(4, 5) should return "Bogey"

golfScore(4, 6) should return "Double Bogey"

golfScore(4, 7) should return "Go Home!"

golfScore(5, 9) should return "Go Home!"

code:

function golfScore(par, strokes) {

// Only change code below this line

if(strokes == 1) {

return "Hole-in-one!";

} else if(strokes <= par-2) {

return "Eagle";

} else if(strokes === par-1) {

return "Birdie";

} else if(strokes === par) {

return "Par";

} else if(strokes === par+1) {

return "Bogey";

} else if(strokes === par+2) {

return "Double Bogey";

} else {

return "Go Home!";

}

// Only change code above this line

}

// Change these values to test

golfScore(4, 7);

**Selecting from many options with Switch Statements**

If you have many options to choose from, use a switchstatement. A switch statement tests a value and can have many case statements which defines various possible values. Statements are executed from the first matched case value until a break is encountered.

Here is a pseudocode example:

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

case values are tested with strict equality (===). The break tells JavaScript to stop executing statements. If the break is omitted, the next statement will be executed.

**Instructions**

Write a switch statement which tests val and sets answer for the following conditions:  
1 - "alpha"  
2 - "beta"  
3 - "gamma"  
4 - "delta"

Here are some helpful links:

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

Code:

Code: 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(1);

Adding a default option in Switch statements

In a switch statement you may not be able to specify all possible values as case statements. Instead, you can add the default statement which will be executed if no matching case statements are found. Think of it like the final else statement in an if/else chain.

A default statement should be the last case.

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

Instructions

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

Code:

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";

break;

}

// Only change code above this line

return answer;

}

// Change this value to test

switchOfStuff("a");

Multiple Identical Options in Switch Statements

If the break statement is omitted from a switchstatement's case, the following case statement(s) are executed until a break is encountered. If you have multiple inputs with the same output, you can represent them in a switch statement 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.

Instructions

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

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

Code:

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;

default:

answer = "None of the above";

break;

}

// Only change code above this line

return answer;

}

// Change this value to test

sequentialSizes(1);

Replacing If Else Chains with Switch

If you have many options to choose from, a switchstatement can be easier to write than many chained if/else if statements. 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";  
}

Instructions

Change the chained if/else if statements into a switch statement.

Code:

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;

case "John":

answer = "";

break;

case 156:

answer = "";

break;

default:

answer = "Oh! my Ghosh!!";

break;

}

// Only change code above this line

return answer;

}

// Change this value to test

chainToSwitch(7);

**Returning Boolean Values from Functions**

You may recall from [**Comparison with the Equality Operator**](https://www.freecodecamp.com/challenges/waypoint-comparison-with-the-equality-operator) that all comparison operators return a boolean true or false value.

A common anti-pattern is to use an if/elsestatement to do a comparison and then returntrue/false:

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

Since === returns true or false, we can return the result of the comparison:

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

**Instructions**

Fix the function isLess to remove the if/elsestatements.

Code:

function isLess(a, b) {

// Fix this code

return a < b;

}

// Change these values to test

isLess(10, 15);

**Return Early Pattern for Functions**

When a return statement 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 return statement.

**Instructions**

Modify the function abTest so that if a or b are less than 0 the function will immediately exit with a value of undefined.

**Hint**  
Remember that [**undefined is a keyword**](http://www.freecodecamp.com/challenges/understanding-uninitialized-variables), not a string.

Code:

// Setup

function abTest(a, b) {

// Only change code below this line

if(a < 0 || 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(2,8);

**Counting Cards**

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 card parameter and increment or decrement the global count variable according to the card's value (see table). The function will then return a string with the current count and the string "Bet" if the count is positive, or "Hold" if the count is zero or negative. The current count and the player's decision ("Bet" or "Hold") should be separated by a single space.

**Example Output**  
"-3 Hold"  
"5 Bet"

**Hint**  
Do NOT reset count to 0 when value is 7, 8, or 9.

Code:

var count = 0;

function cc(card) {

// Only change code below this line

switch(card){

case 2:

case 3:

case 4:

case 5:

case 6:

count += 1;

break;

case 7:

case 8:

case 9:

count += 0;

break;

case 10:

case 'J':

case 'Q':

case 'K':

case 'A':

count -= 1;

break;

}

return count + (count > 0 ? " Bet" : " Hold");

// Only change code above this line

}

// Add/remove calls to test your function.

// Note: Only the last will display

cc(2); cc(3); cc(7); cc('K'); cc('A');

Build JavaScript Objects

You may have heard the term object before.

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.

Here's a sample object:

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

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

Instructions

Make an object that represents a dog called myDog which 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.

Code:

// Example

var ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

// Only change code below this line.

var myDog = {

"name": "Doggy Don",

"legs": 4,

"tails": 1,

"friends": ["Entire dog family!"]

};

Accessing Objects Properties with the Dot Operator

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

The dot operator 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 the dot operator (.) to read an object property:

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

Instructions

Read in the property values of testObj using dot notation. Set the variable hatValue equal to the object property hat and set the variable shirtValue equal to the object property shirt.

Code:

// 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

Accessing Objects Properties with Bracket Notation

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 it, you will need to use bracket notation.

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

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

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

Instructions

Read the values of the properties "an entree" and "the drink" of testObj using bracket notation and assign them to entreeValue and drinkValue respectively.

Code:

// 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

Accessing Objects Properties with Variables

Another use of bracket notation on objects is to use a variable to access a property. This can be very useful for iterating through lists of the object properties or for doing the lookup.

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

var someProp = "propName";  
var myObj = {  
 propName: "Some Value"  
}  
myObj[someProp]; // "Some Value"

Here is one more:

var myDog = "Hunter";  
var dogs = {  
 Fido: "Mutt",

Hunter: "Doberman",

Snoopie: "Beagle"  
}  
var breed = dogs[myDog];  
console.log(breed);// "Doberman"

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*

Instructions

Use the playerNumber variable to lookup player 16 in testObj using bracket notation.

Code:

// Setup

var testObj = {

12: "Namath",

16: "Montana",

19: "Unitas"

};

// Only change code below this line;

var playerNumber = 16; // Change this Line

var player = testObj[playerNumber]; // Change this Line

Updating Object Properties

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".

Instructions

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

Code:

// 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": ["Free Code Camp Campers"]

};

myDog.name = "Happy Coder";

// Only change code below this line.

Add New Properties to a JavaScript Object

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".

Instructions

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

Code:

// 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": ["Free Code Camp Campers"]

};

myDog.bark = "woof";

// Only change code below this line.

Delete Properties from a JavaScript Object

We can also delete properties from objects like this:

delete ourDog.bark;

Instructions

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

Code:

// Example

var ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"],

"bark": "bow-wow"

};

delete ourDog.bark;

// Setup

var myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["Free Code Camp Campers"],

"bark": "woof"

};

delete myDog.tails;

// Only change code below this line.

Using Objects for Lookups

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 switch statement or an if/else chain. 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"

Instructions

Convert the switch statement into a lookup table called lookup. Use it to lookup val and assign the associated string to the result variable.

Code:

// Setup

function phoneticLookup(val) {

// Only change code below this line

var lookup = {

"alpha": "Adams",

"bravo": "Boston",

"charlie": "Chicago",

"delta": "Denver",

"echo": "Easy",

"foxtrot": "Frank",

"": undefined

};

// Only change code above this line

return lookup[val];

}

// Change this value to test

phoneticLookup("charlie");Bottom of Form

Testing Objects for Properties

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 true or false if the property is found or not.

**Example**

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

Instructions

Modify the function checkObj to test myObj for checkProp. If the property is found, return that property's value. If not, return "Not Found".

Code:

// Setup

var myObj = {

gift: "pony",

pet: "kitten",

bed: "sleigh"

};

function checkObj(checkProp) {

// Your Code Here

if (myObj.hasOwnProperty(checkProp) === true){

return myObj[checkProp];

}

return "Not Found";

}

// Test your code by modifying these values

checkObj("gift");

**Manipulating Complex Objects**

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 JSON is 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.

**Instructions**

Add a new album to the myMusic array. Add artist and title strings, release\_year number, and a formatsarray of strings.

Code:

var myMusic = [

{

"artist": "Billy Joel",

"title": "Piano Man",

"release\_year": 1973,

"formats": [

"CS",

"8T",

"LP" ],

"gold": true

},

{

"artist": "Daft Punk",

"title": "Homework",

"release\_year": 1997,

"formats": ["CD", "Cassette", "LP"],

"gold": true

}

];

Accessing Nested Objects

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"

Instructions

Access the myStorage object and assign the contents of the glove box property to the gloveBoxContentsvariable. Use bracket notation for properties with a space in their name.

Code:

// Setup

var myStorage = {

"car": {

"inside": {

"glove box": "maps",

"passenger seat": "crumbs"

},

"outside": {

"trunk": "jack"

}

}

};

// Only change code below this line

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

Accessing Nested Arrays

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"

Instructions

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

Code:

// 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

Iterate with JavaScript For Loops

You can run the same code multiple times by using a loop.

The most common type of JavaScript loop is called a "for loop" because it runs "for" a specific number of times.

For loops are declared with three optional expressions separated by semicolons:

for ([initialization]; [condition]; [final-expression])

The initialization statement is executed one time only before the loop starts. It is typically used to define and setup your loop variable.

The condition statement is evaluated at the beginning of every loop iteration and will continue as long as it evaluates to true. When condition is false at the start of the iteration, the loop will stop executing. This means if condition starts as false, your loop will never execute.

The final-expression is executed at the end of each loop iteration, prior to the next condition check and is usually used to increment or decrement your loop counter.

In the following example we initialize with i = 0 and iterate while our condition i < 5 is true. We'll increment i by 1 in each loop iteration with i++ as our final-expression.

var ourArray = [];  
for (var i = 0; i < 5; i++) {  
 ourArray.push(i);  
}

ourArray will now contain [0,1,2,3,4].

Instructions

Use a for loop to work to push the values 1 through 5 onto myArray.

Code:

// Example

var ourArray = [];

for (var i = 0; i < 5; i++) {

ourArray.push(i);

}

// Setup

var myArray = [];

// Only change code below this line.

for (var i = 1; i < 6; i++) {

myArray.push(i);

}

Iterate Odd Numbers With a For Loop

For loops don't have to iterate one at a time. By changing our final-expression, we can count by even numbers.

We'll start at i = 0 and loop while i < 10. We'll increment i by 2 each loop with i += 2.

var ourArray = [];  
for (var i = 0; i < 10; i += 2) {  
 ourArray.push(i);  
}

ourArray will now contain [0,2,4,6,8].

Let's change our initialization so we can count by odd numbers.

Instructions

Push the odd numbers from 1 through 9 to myArrayusing a for loop.

Code:

// Example

var ourArray = [];

for (var i = 0; i < 10; i += 2) {

ourArray.push(i);

}

// Setup

var myArray = [];

// Only change code below this line.

for (var i =1; i < 10; i += 2) {

myArray.push(i);

}

Count Backwards With a For Loop

A for loop can also count backwards, so long as we can define the right conditions.

In order to count backwards by twos, we'll need to change our initialization, condition, and final-expression.

We'll start at i = 10 and loop while i > 0. We'll decrement i by 2 each loop with i -= 2.

var ourArray = [];  
for (var i=10; i > 0; i-=2) {  
 ourArray.push(i);  
}

ourArray will now contain [10,8,6,4,2].

Let's change our initialization and final-expression so we can count backward by twos by odd numbers.

Instructions

Push the odd numbers from 9 through 1 to myArrayusing a for loop.

Code:

// Example

var ourArray = [];

for (var i = 10; i > 0; i -= 2) {

ourArray.push(i);

}

// Setup

var myArray = [];

// Only change code below this line.

for (var i = 9; i > 0; i -= 2){

myArray.push(i);

}

Iterate Through an Array with a For Loop

A common task in JavaScript is to iterate through the contents of an array. One way to do that is with a forloop. This code will output each element of the array arrto the console:

var arr = [10,9,8,7,6];  
for (var i=0; i < arr.length; i++) {  
 console.log(arr[i]);  
}

Remember that Arrays have zero-based numbering, which means the last index of the array is length - 1. Our *condition* for this loop is i < arr.length, which stops when i is at length - 1.

Instructions

Declare and initialize a variable total to 0. Use a forloop to add the value of each element of the myArr array to total.

Code:

// Example

var ourArr = [ 9, 10, 11, 12];

var ourTotal = 0;

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

ourTotal += ourArr[i];

}

// Setup

var myArr = [ 2, 3, 4, 5, 6];

// Only change code below this line

var total = 0;

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

total += myArr[i];

}

Nesting For Loops

If you have a multi-dimensional array, you can use the same logic as the prior waypoint to loop through both the array and any sub-arrays. Here is an example:

var arr = [  
 [1,2], [3,4], [5,6]  
];  
for (var i=0; i < arr.length; i++) {  
 for (var j=0; j < arr[i].length; j++) {  
 console.log(arr[i][j]);  
 }  
}

This outputs each sub-element in arr one at a time. Note that for the inner loop, we are checking the .length of arr[i], since arr[i] is itself an array.

Instructions

Modify function multiplyAll so that it multiplies the product variable by each number in the sub-arrays of arr

Code:

function multiplyAll(arr) {

var product = 1;

// Only change code below this line

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

for (var j = 0; j < arr[i].length; j++) {

product = product \* arr[i][j];

}

}

// Only change code above this line

return product;

}

// Modify values below to test your code

multiplyAll([[1,2],[3,4],[5,6,7]]);

Iterate with JavaScript While Loops

You can run the same code multiple times by using a loop.

Another type of JavaScript loop is called a "while loop", because it runs "while" a specified condition is true and stops once that condition is no longer true.

var ourArray = [];  
var i = 0;  
while(i < 5) {  
 ourArray.push(i);  
 i++;  
}

Let's try getting a while loop to work by pushing values to an array.

Instructions

Push the numbers 0 through 4 to myArray using a whileloop.

Code:

// Setup

var myArray = [];

// Only change code below this line.

var i = 0;

while(i < 5) {

myArray.push(i);

i++;

}

Profile Lookup

We have an array of objects representing different people in our contacts lists.

A lookUpProfile function that takes firstName and a property (prop) as arguments has been pre-written for you.

The function should check if firstName is an actual contact's firstName and the given property (prop) is a property of that contact.

If both are true, then return the "value" of that property.

If firstName does not correspond to any contacts then return "No such contact"

If prop does not correspond to any valid properties then return "No such property"

Code:

//Setup

var contacts = [

{

"firstName": "Akira",

"lastName": "Laine",

"number": "0543236543",

"likes": ["Pizza", "Coding", "Brownie Points"]

},

{

"firstName": "Harry",

"lastName": "Potter",

"number": "0994372684",

"likes": ["Hogwarts", "Magic", "Hagrid"]

},

{

"firstName": "Sherlock",

"lastName": "Holmes",

"number": "0487345643",

"likes": ["Intriguing Cases", "Violin"]

},

{

"firstName": "Kristian",

"lastName": "Vos",

"number": "unknown",

"likes": ["Javascript", "Gaming", "Foxes"]

}

];

function lookUpProfile(firstName, prop){

// Only change code below this line

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

if (contacts[i].firstName === firstName) {

if(contacts[i].hasOwnProperty(prop)) {

return contacts[i][prop];

} else {

return "No such property";

}

}

}

return "No such contact";

// Only change code above this line

}

// Change these values to test your function

lookUpProfile("Akira", "address");

**Generate Random Fractions with JavaScript**

Random numbers are useful for creating random behavior.

JavaScript has a Math.random() function that generates a random decimal number between 0 (inclusive) and not quite up to 1 (exclusive). Thus Math.random() can return a 0 but never quite return a 1

**Note**  
Like [**Storing Values with the Equal Operator**](https://www.freecodecamp.com/challenges/storing-values-with-the-assignment-operator), all function calls will be resolved before the returnexecutes, so we can return the value of the Math.random() function.

**Instructions**

Change randomFraction to return a random number instead of returning 0.

Code:

function randomFraction() {

// Only change code below this line.

return Math.random();

// Only change code above this line.

}

Generate Random Whole Numbers with JavaScript

It's great that we can generate random decimal numbers, but it's even more useful if we use it to generate random whole numbers.

Use Math.random() to generate a random decimal.

Multiply that random decimal by 20.

Use another function, Math.floor() to round the number down to its nearest whole number.

Remember that Math.random() can never quite return a 1 and, because we're rounding down, it's impossible to actually get 20. This technique will give us a whole number between 0 and 19.

Putting everything together, this is what our code looks like:

Math.floor(Math.random() \* 20);

We are calling Math.random(), multiplying the result by 20, then passing the value to Math.floor() function to round the value down to the nearest whole number.

Instructions

Use this technique to generate and return a random whole number between 0 and 9.

Code:

//var randomNumberBetween0and19 = Math.floor(Math.random() \* 20);

function randomWholeNum() {

// Only change code below this line.

var randomNumberBetween0and9 = Math.floor(Math.random() \* 10);

return randomNumberBetween0and9;

}

Generate Random Whole Numbers within a Range

Instead of generating a random number between zero and a given number like we did before, we can generate a random number that falls within a range of two specific numbers.

To do this, we'll define a minimum number min and a maximum number max.

Here's the formula we'll use. Take a moment to read it and try to understand what this code is doing:

Math.floor(Math.random() \* (max - min + 1)) + min

Instructions

Create a function called randomRange that takes a range myMin and myMax and returns a random number that's greater than or equal to myMin, and is less than or equal to myMax, inclusive.

Code:

// Example

function ourRandomRange(ourMin, ourMax) {

return Math.floor(Math.random() \* (ourMax - ourMin + 1)) + ourMin;

}

ourRandomRange(1, 9);

// Only change code below this line.

function randomRange(myMin, myMax) {

return Math.floor(Math.random() \* (myMax - myMin + 1)) + myMin; // Change this line

}

// Change these values to test your function

var myRandom = randomRange(5, 15);

Sift through Text with Regular Expressions

Regular expressions are used to find certain words or patterns inside of strings.

For example, if we wanted to find the word the in the string The dog chased the cat, we could use the following regular expression: /the/gi

Let's break this down a bit:

/ is the start of the regular expression.

the is the pattern we want to match.

/ is the end of the regular expression.

g means global, which causes the pattern to return all matches in the string, not just the first one.

i means that we want to ignore the case (uppercase or lowercase) when searching for the pattern.

Instructions

Select all the occurrences of the word and in testString.

You can do this by replacing the . part of the regular expression with the word and.

Code:

// Setup

var testString = "Ada Lovelace and Charles Babbage designed the first computer and the software that would have run on it.";

// Example

var expressionToGetSoftware = /software/gi;

var softwareCount = testString.match(expressionToGetSoftware).length;

// Only change code below this line.

var expression = /and/gi; // Change this Line

// Only change code above this line

// This code counts the matches of expression in testString

var andCount = testString.match(expression).length;

Find Numbers with Regular Expressions

We can use special selectors in *Regular Expressions*to select a particular type of value.

One such selector is the digit selector \d which is used to retrieve one digit (e.g. numbers 0 to 9) in a string.

In JavaScript, it is used like this: /\d/g.

Appending a plus sign (+) after the selector, e.g. /\d+/g, allows this regular expression to match one or more digits.

The trailing g is short for 'global', which allows this regular expression to find all matches rather than stop at the first match.

Instructions

Use the \d selector to select the number of numbers in the string, allowing for the possibility of one or more digit.

Code:

// Setup

var testString = "There are 3 cats but 4 dogs.";

// Only change code below this line.

var expression = /\d+/g; // Change this line

// Only change code above this line

// This code counts the matches of expression in testString

var digitCount = testString.match(expression).length;Top of Form

Bottom of Form

Find Whitespace with Regular Expressions

We can also use regular expression selectors like \s to find whitespace in a string.

The whitespace characters are " " (space), \r (the carriage return), \n (newline), \t (tab), and \f (the form feed).

The whitespace regular expression looks like this:

/\s+/g

Instructions

Use \s to select all the whitespace characters in the sentence string.

Code:

// Setup

var testString = "How many spaces are there in this sentence?";

// Only change code below this line.

var expression = /\s+/g; // Change this line

// Only change code above this line

// This code counts the matches of expression in testString

var spaceCount = testString.match(expression).length;

Invert Regular Expression Matches with JavaScript

You can invert any match by using the uppercase version of the regular expression selector.

For example, \s will match any whitespace, and \S will match anything that isn't whitespace.

Instructions

Use /\S/g to count the number of non-whitespace characters in testString.

Code:

// Setup

var testString = "How many non-space characters are there in this sentence?";

// Only change code below this line.

var expression = /\S/g; // Change this line

// Only change code above this line

// This code counts the matches of expression in testString

var nonSpaceCount = testString.match(expression).length;

Declare JavaScript Objects as Variables

Before we dive into Object Oriented Programming, let's revisit JavaScript objects.

Give your motorBike object a wheels, engines and seats attribute and set them to numbers.

Code:

var car = {

"wheels":4,

"engines":1,

"seats":5

};

var motorBike = {

// Only change code below this line.

"wheels": 2,

"engines": 1,

"seats": 1

};

Construct JavaScript Objects with Functions

We are also able to create objects using constructorfunctions.

A constructor function is given a capitalized name to make it clear that it is a constructor.

Here's an example of a constructor function:

var Car = function() {  
  this.wheels = 4;  
  this.engines = 1;  
  this.seats = 5;  
};

In a constructor the this variable refers to the new object being created by the constructor. So when we write,

  this.wheels = 4;

inside of the constructor we are giving the new object it creates a property called wheels with a value of 4.

You can think of a constructor as a description for the object it will create.

Have your MotorBike constructor describe an object with wheels, engines and seats properties and set them to numbers.

Code:

var Car = function() {

this.wheels = 4;

this.engines = 1;

this.seats = 5;

};

// Only change code below this line.

var MotorBike = function() {

this.wheels = 2;

this.engines = 1;

this.seats = 5;

};

Make Instances of Objects with a Constructor Function

Now let's put that great constructor function we made in the last lesson to use!

To use a constructor function we call it with the newkeyword in front of it like:

var myCar = new Car();

myCar is now an instance of the Car constructor that looks like the object it described:

{  
  wheels: 4,  
  engines: 1,  
  seats: 5  
}

Note that it is important to use the new keyword when calling a constructor. This is how Javascript knows to create a new object and that all the references to thisinside the constructor should be referring to this new object.

Now, once the myCar instance is created it can be used like any other object and can have its properties accessed and modified the same way you would usually. For example:

myCar.turboType = "twin";

Our myCar variable now has a property turboType with a value of "twin".

In the editor, use the Car constructor to create a new instance and assign it to myCar.

Then give myCar a nickname property with a string value.

Code:

var Car = function() {

this.wheels = 4;

this.engines = 1;

this.seats = 5;

};

// Only change code below this line.

var myCar = new Car();

myCar.nickname = "Benz";

Make Unique Objects by Passing Parameters to our Constructor

The constructor we have is great, but what if we don't always want to create the same object?

To solve this we can add parameters to our constructor. We do this like the following example:

var Car = function(wheels, seats, engines) {  
  this.wheels = wheels;  
  this.seats = seats;  
  this.engines = engines;  
};

Now we can pass in arguments when we call our constructor.

var myCar = new Car(6, 3, 1);

This code will create an object that uses the argumentswe passed in and looks like:

{  
  wheels: 6,  
  seats: 3,  
  engines: 1  
}

Now give it a try yourself! Alter the Car constructor to use parameters to assign values to the wheels, seats, and engines properties.

Then call your new constructor with three number arguments and assign it to myCar to see it in action.

Top of Form

code:

var Car = function(wheels, seats, engines) {

//Change this constructor

this.wheels = wheels;

this.seats = seats;

this.engines = engines;

};

//Try it out here

var myCar = new Car(4, 5, 1);

**Make Object Properties Private**

Objects have their own attributes, called properties, and their own functions, called methods.

In the [**previous challenges**](https://www.freecodecamp.com/challenges/make-instances-of-objects-with-a-constructor-function), we used the this keyword to reference public properties of the current object.

We can also create private properties and private methods, which aren't accessible from outside the object.

To do this, we create the variable inside the constructorusing the var keyword we're familiar with, instead of creating it as a property of this.

This is useful for when we need to store information about an object but we want to control how it is used by outside code.

For example, what if we want to store the speed our car is traveling at but we only want outside code to be able to modify it by accelerating or decelerating, so the speed changes in a controlled way?

In the editor you can see an example of a Carconstructor that implements this pattern.

Now try it yourself! Modify the Bike constructor to have a private property called gear and two public methods called getGear and setGear to get and set that value.

[**Further explanation on this keyword**](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/this)

Code:

var Car = function() {

// this is a private variable

var speed = 10;

// these are public methods

this.accelerate = function(change) {

speed += change;

};

this.decelerate = function() {

speed -= 5;

};

this.getSpeed = function() {

return speed;

};

};

var Bike = function() {

// Only change code below this line.

var gear = 0;

this.getGear = function() {

return gear;

};

this.setGear = function(set) {

gear = set;

};

};

var myCar = new Car();

var myBike = new Bike();

myBike.setGear(3);

Iterate over Arrays with map

The map method is a convenient way to iterate through arrays. Here's an example usage:

var oldArray = [1, 2, 3];  
var timesFour = oldArray.map(function(val){  
  return val \* 4;  
});  
console.log(timesFour); // returns [4, 8, 12]  
console.log(oldArray); // returns [1, 2, 3]

The map method will iterate through every element of the array, creating a new array with values that have been modified by the callback function, and return it. Note that it does not modify the original array.

In our example the callback only uses the value of the array element (the val argument) but your callback can also include arguments for the index and array being acted on.

Use the map function to add 3 to every value in the variable oldArray, and save the results into variable newArray. oldArray should not change.

Code:

var oldArray = [1,2,3,4,5];

// Only change code below this line.

var newArray = oldArray.map(function(val) {

return val + 3;

});

Condense arrays with reduce

The array method reduce is used to iterate through an array and condense it into one value.

To use reduce you pass in a callback whose arguments are an accumulator (in this case, previousVal) and the current value (currentVal).

The accumulator is like a total that reduce keeps track of after each operation. The current value is just the next element in the array you're iterating through.

reduce has an optional second argument which can be used to set the initial value of the accumulator. If no initial value is specified it will be the first array element and currentVal will start with the second array element.

Here is an example of reduce being used to subtract all the values of an array:

var singleVal = array.reduce(function(previousVal, currentVal) {  
  return previousVal - currentVal;  
}, 0);

Use the reduce method to sum all the values in arrayand assign it to singleVal.

Code:

var array = [4,5,6,7,8];

var singleVal = 0;

// Only change code below this line.

singleVal = array.reduce(function(previousVal, currentVal) {

return previousVal + currentVal;

});

Filter Arrays with filter

The filter method is used to iterate through an array and filter out elements where a given condition is not true.

filter is passed a callback function which takes the current value (we've called that val) as an argument.

Any array element for which the callback returns true will be kept and elements that return false will be filtered out.

The following code is an example of using filter to remove array elements that are equal to five:

Note: We omit the second and third arguments since we only need the value

array = array.filter(function(val) {  
  return val !== 5;  
});

Use filter to create a new array with all the values from oldArray which are less than 6. The oldArrayshould not change.

Code:

var oldArray = [1,2,3,4,5,6,7,8,9,10];

// Only change code below this line.

var newArray = oldArray.filter(function(val) {

return val < 6;

});

**Sort Arrays with sort**

You can use the method sort to easily sort the values in an array alphabetically or numerically.

Unlike the previous array methods we have been looking at, sort actually alters the array in place. However, it also returns this sorted array.

sort can be passed a compare function as a callback. The compare function should return a negative number if ashould be before b, a positive number if a should be after b, or 0 if they are equal.

If no compare (callback) function is passed in, it will convert the values to strings and sort alphabetically.

Here is an example of using sort with a compare function that will sort the elements from smallest to largest number:

var array = [1, 12, 21, 2];  
array.sort(function(a, b) {  
  return a - b;  
});

Use sort to sort array from largest to smallest.

Further explanation on the .sort() method can be found [**here**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort).

Code:

var array = [1, 12, 21, 2];

// Only change code below this line.

array.sort(function(a, b) {

return b - a;

});

Reverse Arrays with reverse

You can use the reverse method to reverse the elements of an array.

reverse is another array method that alters the array in place, but it also returns the reversed array.

var myArray = [1, 2, 3];  
myArray.reverse();

*returns [3, 2, 1]*

Use reverse to reverse the array variable and assign it to newArray.

Code:

var array = [1,2,3,4,5,6,7];

var newArray = [];

// Only change code below this line.

newArray = array.reverse();

Concatenate Arrays with concat

concat can be used to merge the contents of two arrays into one.

concat takes an array as an argument and returns a new array with the elements of this array concatenated onto the end.

Here is an example of concat being used to concatenate otherArray onto the end of oldArray:

newArray = oldArray.concat(otherArray);

Use .concat() to concatenate concatMe onto the end of oldArray and assign it to newArray.

Code:

var oldArray = [1,2,3];

var newArray = [];

var concatMe = [4,5,6];

// Only change code below this line.

newArray = oldArray.concat(concatMe);

Split Strings with split

You can use the split method to split a string into an array.

split uses the argument you pass in as a delimiter to determine which points the string should be split at.

Here is an example of split being used to split a string at every s character:

var array = string.split('s');

Use split to create an array of words from string and assign it to array.

Code:

var string = "Split me into an array";

var array = [];

// Only change code below this line.

array = string.split(' ');

Join Strings with join

We can use the join method to join each element of an array into a string separated by whatever delimiter you provide as an argument.

The following is an example of using join to join all of the elements of an array into a string with all the elements separated by word and:

var veggies = ["Celery", "Radish", "Carrot", "Potato"];  
var salad = veggies.join(" and ");  
console.log(salad); // "Celery and Radish and Carrot and Potato"

Use the join method to create a string from joinMewith spaces in between each element and assign it to joinedString.

Code:

var joinMe = ["Split","me","into","an","array"];

var joinedString = '';

// Only change code below this line.

joinedString = joinMe.join(' ');

Bottom of Form