**Basic JavaScript**

Comment your JavaScript Code 2

Declare JavaScript Variables 3

Storing Values with the Assignment Operator 3

Initializing Variables with the Assignment Operator 3

Understanding Uninitialized Variables 3

Understanding Case Sensitivity in Variables 4

Mathematical Computation: 4

Increment a Number with JavaScript 5

Decrement a Number with JavaScript 5

Create Decimal Numbers with JavaScript 5

Compound Assignment With Augmented Addition, Subtraction, Multip., Divi. 5

Convert Celsius to Fahrenheit 6

Declare String Variables 6

Escaping Literal Quotes in Strings 6

Quoting Strings with Single Quotes 6

Escape Sequences in Strings 6

Concatenating Strings with Plus Operator 7

Concatenating Strings with the Plus Equals Operator 7

Constructing Strings with Variables 7

Appending Variables to Strings 7

Find the Length of a String 7

Use Bracket Notation to Find the First Character in a String 7

Understand String Immutability 8

Use Bracket Notation to Find the Last Character in a String 8

Use Bracket Notation to Find the NthtoLast Character in a String 8

Word Blanks (Arrays) 8

Store Multiple Values in one Variable using JavaScript Arrays 9

Access & Modify Array Data with Indexes 9

Access MultiDimensional Arrays With Indexes 9

Manipulate Arrays With push 9

Manipulate Arrays With pop 10

Manipulate Arrays With shift 10

Manipulate Arrays With unshift 10

Write Reusable JavaScript with Functions 10

Passing Values to Functions with Arguments 11

Global Scope and Functions 11

Local Scope and Functions 11

Global vs Local Scope in Functions 11

Return a Value from a Function with Return 11

Assignment with a Returned Value 12

Stand In Line 12

Use Conditional Logic with If Statements 12

Comparison with the Equality Operator and Strict Equality Operator 13

Comparison with the Inequality Operator & Strict Inequality Operator 14

Comparison with the Greater/Less Than/ Equal to Operator 14

Comparisons with the Logical And Operator 15

Comparisons with the Logical Or Operator 15

Introducing Else Statements 15

Introducing Else If Statements 15

Selecting from many options with Switch Statements 16

Adding a default option in Switch statements 17

Multiple Identical Options in Switch Statements 17

Replacing If Else Chains with Switch 18

Returning Boolean Values from Functions 18

Counting Cards 19

Building JavaScript Objects 19

Accessing Objects Properties with the Dot Operator 20

Accessing Objects Properties with Bracket Notation 20

Accessing Objects Properties with Variables 20

Updating Object Properties 21

Add & Delete New Properties to a JavaScript Object 21

Using Objects for Lookups 22

Testing Objects for Properties 23

Manipulating Complex Objects 24

Accessing Nested Objects 25

Accessing Nested Arrays 26

Record Collection 27

Loops 28

Iterate with JavaScript For Loops 28

Iterate Odd Numbers With a For Loop 28

Count Backwards With a For Loop 29

Iterate Through an Array with a For Loop 29

Nesting For Loops 30

Iterate with JavaScript While Loops 30

Profile Lookup 31

Generating Random Values 32

Generate Random Fractions with JavaScript 32

Generate Random Whole Numbers with JavaScript 32

Generate Random Whole Numbers within a Range 32

Finding & Sifting Regular Expressions 33

Sift through Text with Regular Expressions 33

Find Numbers with Regular Expressions 33

Find Whitespace with Regular Expressions 33

Invert Regular Expression Matches with JavaScript 34

## Comment your JavaScript Code

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.

## Declare JavaScript Variables

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.

## 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 5 again and assigns it to myNum.

## 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.

## Understanding Uninitialized Variables

When JavaScript variables are declared, they have an initial value of undefined. If you do a mathematical operation on anundefined 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".

// Initialize these three variables

var a;

var b;

var c;

// Do not change code below this line

a = a + 1;

b = b + 5;

c = c + " String!";

**Answer:**

// 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;

# Mathematical Computation:

* Addition

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

* Subtraction

JavaScript uses the - symbol for subtraction.

**Example** myVar = 12 - 6; // assigned 6

* Multiplication

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

**Example** myVar = 13 \* 13; // assigned 169

* Division

JavaScript uses the / symbol for division.

**Example** myVar = 16 / 2; // assigned 8

* Remainders; Modulus

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

**Example**

5 % 2 = 1

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.

## 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.

## 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.

## 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" \l "Accuracy_problems" \t "_blank)**.

## Compound Assignment With Augmented Addition, Subtraction, Multip., Divi.

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.

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

myVar = myVar - 5;

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

myVar -= 5;

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

myVar = myVar \* 5;

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

myVar \*= 5;

* The /= operator divides a variable by another number.

myVar = myVar / 5;

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

myVar /= 5;

## Convert Celsius to Fahrenheit

function convertToF(celsius) {

var fahrenheit;

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

return fahrenheit;

}

convertToF(30);

# 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.

## 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".

## 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.'

## 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  *Note that the backslash itself must be escaped in order to display as a backslash.* |
| \f | form feed |

## Concatenating Strings with Plus Operator

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

**Example**

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

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

## 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.

## Constructing Strings with Variables

Sometimes you will need to build a string, **[Mad Libs](https://en.wikipedia.org/wiki/Mad_Libs" \t "_blank)** style. By using the concatenation operator (+), you can insert one or more variables into a string you're building.

var ourName = "Free Code Camp";

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

## 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.

var anAdjective = "awesome!";

var ourStr = "Free Code Camp is ";

ourStr += anAdjective;

## Find the Length of a String

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

"Alan Peter".length; // 10

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

var firstNameLength = 0;

var firstName = "Ada";

firstNameLength = firstName.length;

## Use Bracket Notation to Find the First Character in a String

Bracket notation is a way to get a character at a specificindex 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].

var firstLetterOfFirstName = "";

var firstName = "Ada";

firstLetterOfFirstName = firstName[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";

## 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 usingfirstName[firstName.length - 1].

var firstName = "Ada";

var lastLetterOfFirstName = firstName[firstName.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 usingfirstName[firstName.length - 3]

var firstName = "Ada";

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

# Word Blanks (Arrays)

We will now use our knowledge of strings to build a "**[Mad Libs](https://en.wikipedia.org/wiki/Mad_Libs" \t "_blank)**" 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.

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

var result = "";

// Your code below this line

result = "My " + myNoun + " " + myAdjective + " " + myVerb + " " + myAdverb;

// Your code above this line

return result;

}

// Change the words here to test your function

wordBlanks("dog", "big", "ran", "quickly");

Someone mentioned this also would work:

result = My ${myNoun} ${myAdjective} ${myVerb} ${myAdverb}

See this for more info: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals>

## 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"].

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

## Access & Modify 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

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]

## 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

## Manipulate Arrays With push

An easy way to append data to the end of an array is via thepush() 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]

## 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].

var ourArray = [1,2,3];

var removedFromOurArray = ourArray.pop();

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

## 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.

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

removedFromOurArray = ourArray.shift();

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

## 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.

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

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

ourArray.unshift("Happy");

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

# 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.

function ourReusableFunction() {

console.log("Heyya, World");

}

ourReusableFunction();

## 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 param2will equal "World". Note that you could call testFun again with different arguments and the parameters would take on the value of the new arguments.

function ourFunctionWithArgs(a, b) {

console.log(a - b);

}

ourFunctionWithArgs(10, 5); // Outputs 5

## 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.

## 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.

## 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 local variable 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 localversion of the variable is present.

## 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.

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.

## 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" \t "_blank)**, 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.

var changed = 0;

function change(num) {

return (num + 5) / 3;

}

changed = change(10);

## 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. ThenextInLine function should then return the element that was removed.

function nextInLine(arr, item) {

// Your code here

arr.push(item);

var remove = arr.shift();

return remove; // 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));

## 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 Boolean conditions 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 if statement evaluates myCondition to see if it is true or not. Since it istrue, the function returns "It was true". When we calltest 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".

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 and Strict Equality Operator

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

The most basic operator is the equality operator ==. The equality operator compares two values and returns true if 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

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 aString type.

## Comparison with the Inequality Operator & Strict 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

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 true and*vice versa*. Strict inequality will not convert data types.

**Examples**

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

## Comparison with the Greater/Less Than/ Equal to 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

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 to operator will convert data types while comparing.

**Examples**

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

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

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 returnsfalse. Like the equality operator, less than or equal toconverts data types.

**Examples**

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

## 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";

## 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";

# 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 elsestatement, an alternate block of code can be executed.

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

## Introducing Else If Statements

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

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

## 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 breaktells 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 answerfor the following conditions:  
1 - "alpha"  
2 - "beta"  
3 - "gamma"  
4 - "delta"

function caseInSwitch(val) {

var answer = "";

// Only change code below this line

switch (val){

case 1:

answer = "alpha";

break;

case 2:

answer = "beta";

break;

case 3:

answer = "gamma";

break;

case 4:

answer = "delta";

break;

}

// Only change code above this line

return answer;

}

// Change this value to test

caseInSwitch(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 thedefault statement which will be executed if no matchingcase statements are found. Think of it like the final elsestatement 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;  
}

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(1);

## Multiple Identical Options in Switch Statements

If the break statement is omitted from a switch statement'scase, the following case statement(s) are executed until abreak 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.

## Replacing If Else Chains with Switch

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

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

can be replaced with:

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

## 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" \t "_blank)**that all comparison operators return a boolean true orfalse value.

A common anti-pattern is to use an if/else statement to do a comparison and then return true/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;  
}

## Counting Cards

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 = count + 1;

break;

case 7:

case 8:

case 9:

count = count;

break;

case 10:

case "J":

case "Q":

case "K":

case "A":

count = count - 1;

}

if (count > 0) {

return count + " Bet";

} else {

return count + " Hold";

}

return "Change Me";

// 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');

Cards Sequence 2, 3, 4, 5, 6 should return"5 Bet"

Cards Sequence 7, 8, 9 should return "0 Hold"

Cards Sequence 10, J, Q, K, A should return"-5 Hold"

Cards Sequence 3, 7, Q, 8, A should return"-1 Hold"

Cards Sequence 2, J, 9, 2, 7 should return"1 Bet"

Cards Sequence 2, 2, 10 should return "1 Bet"

Cards Sequence 3, 2, A, 10, K should return"-1 Hold"

# Building 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.

## 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

## 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).

// 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*

// 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".

## Add & Delete 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.

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";

We can also delete properties from objects like this:

delete ourDog.bark;

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"

};

// Only change code below this line.

delete myDog.tails;

## 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 calledlookup. Use it to lookup val and assign the associated string to the result variable.

BEFORE

// Setup

function phoneticLookup(val) {

var result = "";

// Only change code below this line

switch(val) {

case "alpha":

result = "Adams";

break;

case "bravo":

result = "Boston";

break;

case "charlie":

result = "Chicago";

break;

case "delta":

result = "Denver";

break;

case "echo":

result = "Easy";

break;

case "foxtrot":

result = "Frank";

}

// Only change code above this line

return result;

}

// Change this value to test

phoneticLookup("charlie");

AFTER

// Setup

function phoneticLookup(val) {

var result = "";

// Only change code below this line

var lookup = {

"alpha": "Adams",

"bravo": "Boston",

"charlie": "Chicago",

"delta": "Denver",

"echo": "Easy",

"foxtrot": "Frank"

};

result = lookup[val];

// Only change code above this line

return result;

}

// Change this value to test

phoneticLookup("charlie");

## 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 orfalse if the property is found or not.

**Example**

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

// Setup

var myObj = {

gift: "pony",

pet: "kitten",

bed: "sleigh"

};

function checkObj(checkProp) {

// Your Code Here

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

return myObj[checkProp];

}

else{

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/" \t "_blank)** 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.

var myMusic = [

{

"artist": "Billy Joel",

"title": "Piano Man",

"release\_year": 1973,

"formats": [

"CS",

"8T",

"LP" ],

"gold": true

},

{

"artist": "The Beatles",

"title": "I'm Looking Through You",

"release\_year": 1965,

"formats": [

"CD",

"Vinyl"]

}

// Add record here

];

## 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"

// 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"

// Setup

var myPlants = [

{

type: "flowers",

list: [

"rose",

"tulip",

"dandelion"

]

},

{

type: "trees",

list: [

"fir",

"pine",

"birch"

]

}

];

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

## Record Collection

You are given a JSON object representing a part of your musical album collection. Each album has several properties and a unique id number as its key. Not all albums have complete information.

Write a function which takes an album's id (like 2548), a property prop (like "artist" or "tracks"), and a value(like "Addicted to Love") to modify the data in this collection.

If prop isn't "tracks" and value isn't empty (""), update or set the value for that record album's property.

Your function must always return the entire collection object.

There are several rules for handling incomplete data:

If prop is "tracks" but the album doesn't have a "tracks"property, create an empty array before adding the new value to the album's corresponding property.

If prop is "tracks" and value isn't empty (""), push thevalue onto the end of the album's existing tracks array.

If value is empty (""), delete the given prop property from the album.

**Hints**  
Use bracket notation when **[accessing object properties with variables](https://www.freecodecamp.com/challenges/accessing-objects-properties-with-variables" \t "_blank)**.

Push is an array method you can read about on [**Mozilla Developer Network**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push).

You may refer back to [**Manipulating Complex Objects**](https://www.freecodecamp.com/challenges/manipulating-complex-objects)Introducing JavaScript Object Notation (JSON) for a refresher.

// Setup

var collection = {

"2548": {

"album": "Slippery When Wet",

"artist": "Bon Jovi",

"tracks": [

"Let It Rock",

"You Give Love a Bad Name"

]

},

"2468": {

"album": "1999",

"artist": "Prince",

"tracks": [

"1999",

"Little Red Corvette"

]

},

"1245": {

"artist": "Robert Palmer",

"tracks": [ ]

},

"5439": {

"album": "ABBA Gold"

}

};

// Keep a copy of the collection for tests

var collectionCopy = JSON.parse(JSON.stringify(collection));

// Only change code below this line

function updateRecords(id, prop, value) {

if (value !== "" && prop == "tracks") {

if (collection[id].hasOwnProperty(prop) === true){

(collection[id][prop]).push(value);

}

else {

collection[id][prop] = [value];

}

}

else if (value !== "" && prop !== "tracks") {

collection[id][prop] = value;

}

else if (value === "") {

delete collection[id][prop];

}

return collection;

}

// Alter values below to test your code

updateRecords(5439, "artist", "ABBA");

# Loops

## 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 totrue. When condition is false at the start of the iteration, the loop will stop executing. This means ifcondition 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].

var ourArray = [];

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

ourArray.push(i);

}

// Setup

var myArray = [];

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 ourfinal-expression, we can count by even numbers.

We'll start at i = 0 and loop while i < 10. We'll incrementi 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.

// Example

var ourArray = [];

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

ourArray.push(i);

}

// Setup

var myArray = [];

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 ourinitialization, condition, and final-expression.

We'll start at i = 10 and loop while i > 0. We'll decrementi 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-expressionso we can count backward by twos by odd numbers.

#### Instructions

Push the odd numbers from 9 through 1 to myArray using afor loop.

var ourArray = [];

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

ourArray.push(i);

}

// Setup

var myArray = [];

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 for loop. This code will output each element of the array arr to 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 conditionfor 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 for loop to add the value of each element of the myArr array tototal.

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];

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 theproduct variable by each number in the sub-arrays of arr

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

console.log(product);

}

}

// 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.

// 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'sfirstName 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"

//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

i = 0;

while (i < contacts.length){

if (contacts[i].firstName === firstName && contacts[i].hasOwnProperty(prop) === true){

return contacts[i][prop];

} else if (contacts[i].firstName === firstName && contacts[i].hasOwnProperty(prop) === false){

return "No such property";

} else{

i = i + 1;}

}

return "No such contact";

// Only change code above this line

}

OTHER SUGGESTED ANSWER:

function lookUpProfile(firstName, prop){

// Only change code below this line

i = 0;

while (i < contacts.length){

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

{

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

{

return contacts[i][prop];

}

else

{

return "No such property";

}

}

i = i + 1;

}

return "No such contact";

// Only change code above this line

}

# Generating Random Values

## 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" \t "_blank)**, all function calls will be resolved before the return executes, so we canreturn the value of the Math.random() function.

## 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.

1. Use Math.random() to generate a random decimal.
2. Multiply that random decimal by 20.
3. Use another function, Math.floor() to round the number down to its nearest whole number.

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

## 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

# Finding & Sifting Regular Expressions

## 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 stringThe dog chased the cat, we could use the followingregular 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.

// Setup

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

var expressionToGetSoftware = /software/gi;

var softwareCount = testString.match(expressionToGetSoftware).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.

// 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;

## 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.

// 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.

// 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;