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.

#### 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]

#### 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].

#### 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.

#### 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.

#### 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.

#### 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 globalvariables with the same name. When you do this, the local variable takes precedence over the globalvariable.

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.

#### 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.

#### 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 17and assigns it to ourSum variable.

#### 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 trueor false.

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

**Pseudocode**

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

**Example**

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

When 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".

#### 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;  
}

#### 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.

#### 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.

#### 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).

#### 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

#### 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 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".

#### Delete Properties from a JavaScript Object

We can also delete properties from objects like this:

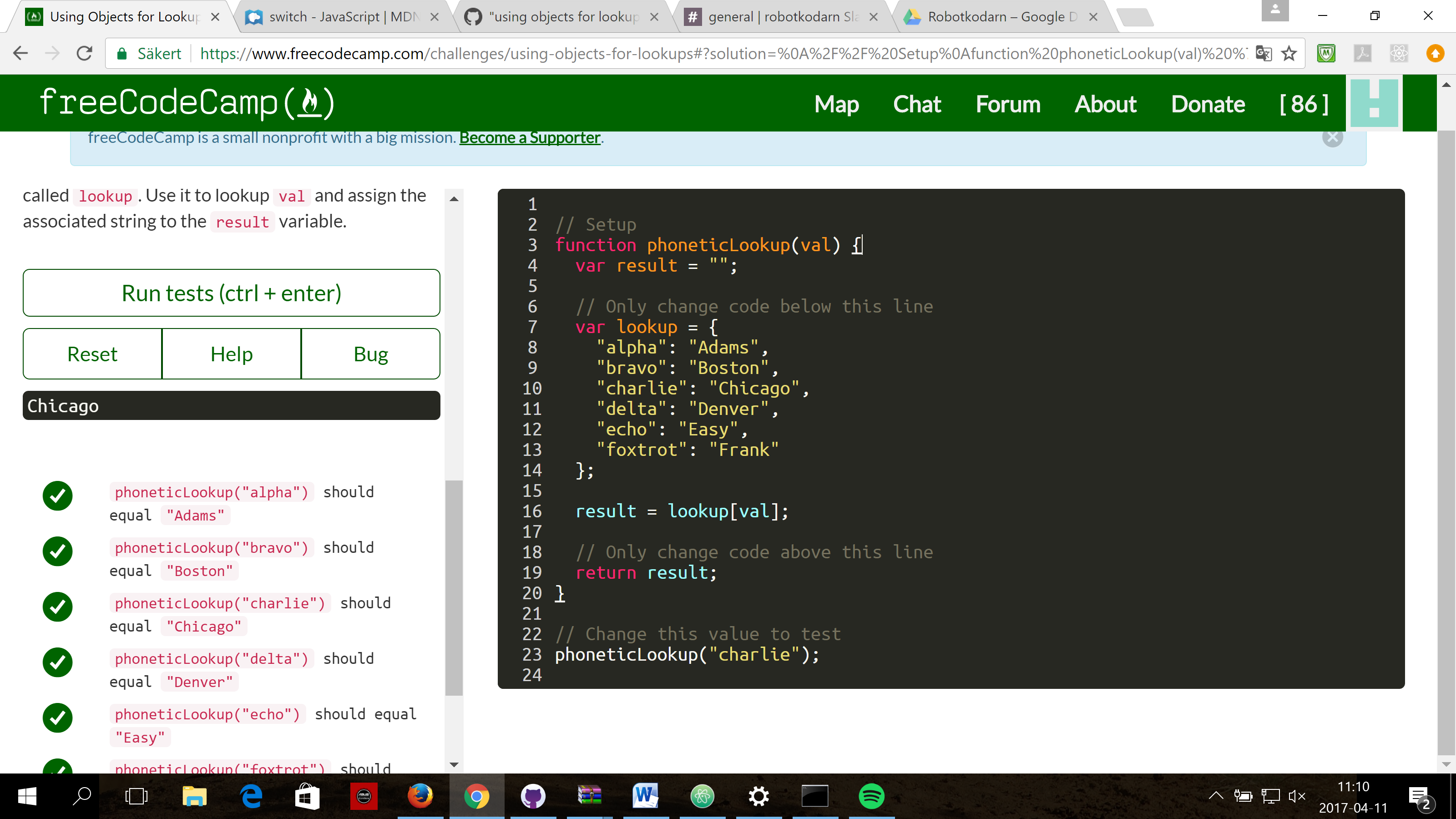
delete ourDog.bark;

#### 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 switchstatement 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"

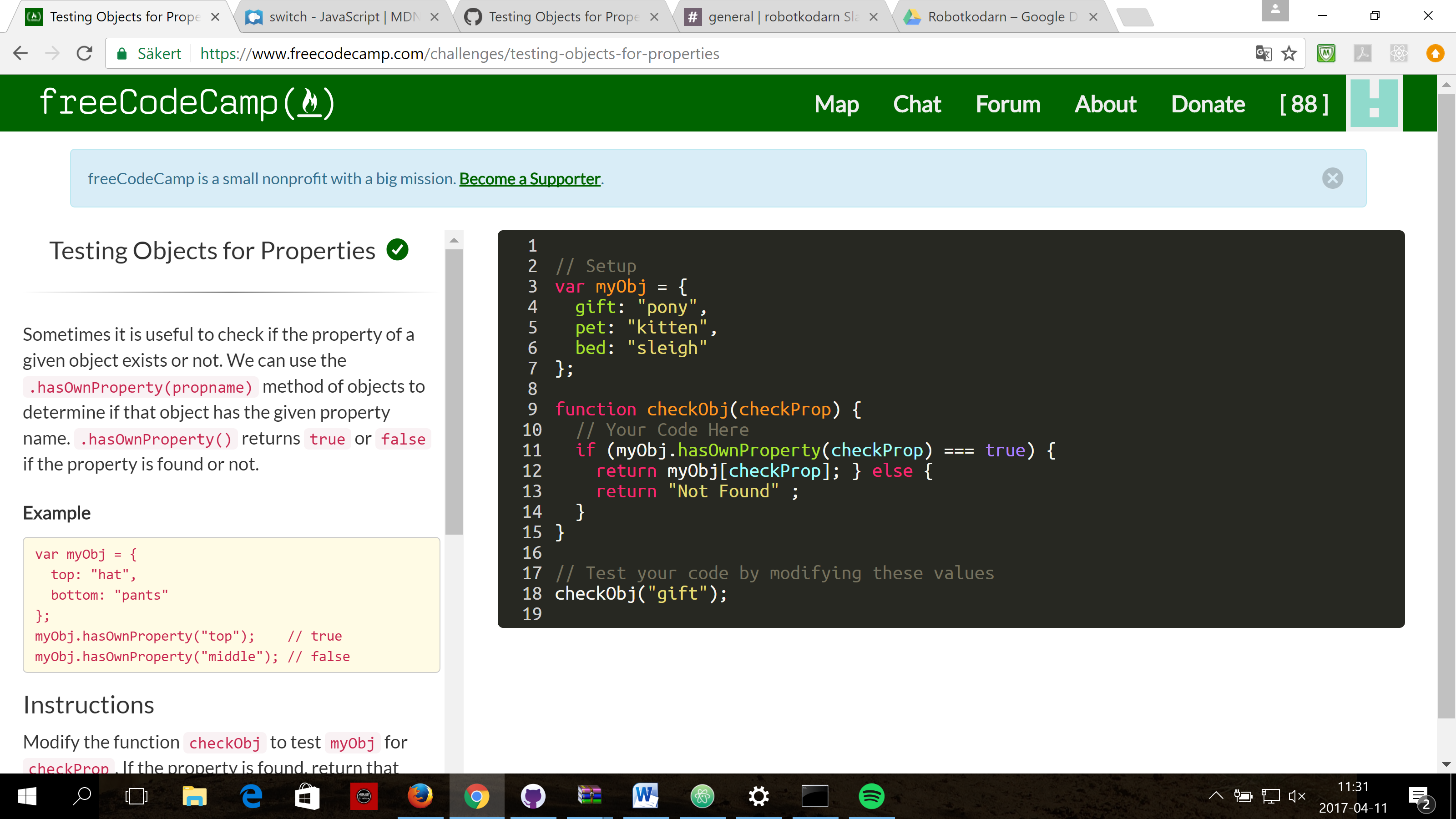


#### Testing Objects for Properties

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 falseif the property is found or not.

**Example**

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



#### 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.

#### 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"

#### Use bracket notation for properties with a space in their name.

#### 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"

#### 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 conditioncheck and is usually used to increment or decrement your loop counter.

In the following example we initialize with i = 0and 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].

#### 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.

#### 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].

#### 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 condition for this loop is i < arr.length, which stops when i is at length - 1.