JavaScript Notes
Susie Chhuor
Last edit 03/10/2018

HTML - CSS - JS

- JavaScript **computes** values and adds **behavior** into your page
    - Made up of statements
- HTML **declares** the **structure** and **content** of page
- CSS is the **style** of your page

How To Add JS to Web Pages	

Inline	Adding into your HTML5 files
	
	`<scripts type = 'text/javascript'>`
	`        {code block}`
	`</script>`

External	Linking into your HTML5 file
	
	`<script src = 'myJavsScript.js'>`
	** do not insert code block. **
	`</script>`

Notes	Cannot use both inline and external codes together in the **same** script
	
	WRONG:
	<script **src** = 'myJavsScript.js'> ← external link
	var greeting = 'hello' ; ← inline text
	</script>
	
	If you want to use inline + external it must be in different scripts.
	
	CORRECT:
	`<script src = 'myJavsScript.js'> </script>`
	
	`<script type = 'text/javascript'>`
	`      var greeting = 'hello';`
	`</script>`

Data Types	
Null	Object doesn't exist and needs to be created or can be skipped Check to see if object exists: `if (weather != null) {}` You can also check check for null, using == , but it is not advised. Instead use other

	methods, such as:
	`weather === undefined`
	`!weather`
	`!(key in objectName)`

NaN	Stands for 'Not a Number' that is a number (type), but cannot be represented properly on computer Testing for NaN requires a function because `NaN != NaN (they are not equal)` so we use: If (`isNan(myNum)`) { code } If it returns TRUE it passes the code 0/0 = NaN 'Food' * 1000 → NaN
Undefined	Variable or object has no value

Equality & Comparison Operators	
=	Assigns a value to a variable Example: Var greeting = 'hello' 'hello' is assigned to the variable greeting
==	Equality Compares one value to another, *but they do not need to be same type* Example: Var testMe = '99' ← this is a string If (testMe == 99) { code block } ← this is a integer Is this true? Yes, because it compares the values and not the type. 99 == '99' → TRUE because **JS converts the string to a number**. 99 == 99 **RULES:** 1 == true 0 == false undefined == null → true " " == 0 → empty strings are 0
!=	checks for inequalities, but will still convert strings to numbers if necessary. Type doesn't matter. **Example:** Nan != Nan → true

<= >=	Less than or equal to Greater than or equal to Only compares **strings || numbers** **Example:** 99 <= '100' → TRUE The string, '100' is converted to the number 100 because **JS converts the string to a number** <= && >= follows == rules
===	Strict equality / identity operator Strictly equal if and only if they are the same **type** and same **value**
!==	' not equal to ' is a strict comparison → **value** && **types** must match.
Notes	MDN Reference

Comparisons	
comparing booleans and number	0 < true true is converted to 1 this is TRUE because 0 < 1 5 > 5 When the same number is evaluated to one another → FALSE 5 >= 5 If you want the same number to evaluate true, you must have **=**
comparing string and string	'banana' < 'mango' TRUE because the first letter 'b' < 'm' **What if string has capitalization?** 'Mango' < 'mango' → TRUE because regardless of capitalization, JS follows keycodes which assign numerical values. 'M' = 77 'm' = 109 77 < 109

Logical Operators	
| |	OR ● Results TRUE if **either** of the two expressions is true u
&&	AND ● Results in TRUE if **both** of two expressions is true

!	means ‘not’ ● Results in TRUE **if** the expression is *False* **Example:** While (string[i] !== undefined) → the code block WILL run if the string is NOT undefined. The string[i] has a value.

Concatenation	

Concatenation with
+ Operator with numbers
	When using + operator <u>with numbers only</u>, you get addition
	`var add = 3 + 3`
	`add = 6`

Concatenation with Strings only	When using + operator <u>with strings only</u>, you get concatenation
	`var greeting = ‘hello’ + ‘Susie’`
	`greeting = ‘hello Susie’`

Concatenation with Numbers AND Strings
 	
Other Arithmetic operators w/
Numbers AND Strings
	When using + operator with <u>strings & numbers</u> :
	● JS will convert the **number to a string**
	Example:
	`var add = 3 + ‘4’`
	`var add = ‘3’ + ‘4’`
	`var add = ‘34’`
	● To convert a **string to a number**
	use function Number
	`var num = 3 + Number(‘4’)`
	`var num = 3 + 4`
	`var num = 7`
	
	When using == it converts **strings to numbers** - this is the opposite!
	
	When using OTHER arithmetic operator with <u>strings & numbers</u> :
	● JS will just do the math. No number to string conversions or concatenation
	`var multiply = 3 * ‘4’`
	`var multiply = 3*4`
	`var multiply = 12`

5 Falsey Values Values that aren’t exactly true or false, but behave like so in conditional statements (if)	**Truthy Examples** If not falsey, then it’s truthy
undefined	[ ] → empty arrays
null	{ }
0	1
empty String → ‘ ‘	Strings are truthy because only empty strings

	are falsey. Var string = ‘hello’ If (string) { code block; } →
NaN	

LOOPS	
FOR LOOPS	**WHILE LOOPS**

function compareLoops(stuff){
 	
for (var j = 0; j < 10; j++) {
console.log('jason is cool');
}
	function compareLoops(stuff){
	
	var i = 0;
	while (i < 10){
	i++;
	console.log('susie is cool');
	}

Syntax
For (initial statement, conditional statement, increment statement) {code block }
	Syntax
	
	Initial statement (declares variable)
	While (conditional statement) {
	Increment statement;
	}

● Use with Arrays ● Use with objects	● ** variable should be declared before the while loop so that it is defined since code reads from top to bottom ● Used when you don’t know how many times you need to loop ● Looping until the conditional statement is met
FOR IN LOOPS	**RECURSION**
for (var key in objName) { }	related to loops
Syntax	
Iterates through every key in an object You can access the key using bracket notations	
CHEAT SHEET	

Checking Types	
String	`typeof valueBeingChecked`
Array	`Array.isArray(valueBeingChecked)`
Object	
Boolean	

FUNCTIONS

Are codes that can be **reused** over and over again.

Allows you to reuse the code as many times as you like by using **different variables in the parameters** to yield a result/output without having to rewrite the code multiply times.

Syntax of Function

```
function bark(name, weight) {

      code block
      Return __ ;
}

Bark(name,weight);
```

Reading the Function

1. Begin a function w/ 'function'
2. Give the function a name (camelCase)
3. Functions have **parameters**, which are the changing variables
   Each value you pass is assigned to a corresponding parameter
4. Code block goes outside { }
5. Return (optional)
   Returns whatever is in the code block
   If there is a function without a return statement → returns **undefined**
6. **Call / Invoke** the function by calling its **name**
7. Arguments are another name for values you pass into function when you CALL / INVOKE
   Arguments allow you to customize the code - different arguments yields different results
   Functions are parameterized → each time you use function, you pass in arguments

Parameters and Arguments are Different

You define a **parameter** *once*, but call **arguments** multiple times.

What HAPPENS?
Call a function → pass in arguments → **arguments** are copied into **parameters** in the **function definition**

Bark(**'Toki', 15**); → pass by value matches → **function bark**(**name,weight**)

Function Argument & Parameter Cases			

Practice Function
`function makeTea(cups, tea) {`
`    console.log('Brewing ' + cups + ' cups of ' + tea);`
`}`			

Sample Case	**Calling the function**	**Results**	**JavaScript Console**
Not enough arguments to pass through all parameters	makeTea(3);	Each parameter that doesn't have a matching argument will return **undefined**	Brewing 3 cups of **undefined**
Passing too many arguments (more than parameters)	makeTea(3, 'Earl Grey', 'hey!', 42);	JS ignores the extra arguments	Brewing 3 cups of Earl Grey

Practice Function
`function barkAtMoon() {`
`    console.log('Wooooooo!');`
`}`			

No parameters	Nothing, not all functions have parameters	Whatever is in the code block	Wooooooo!

VARIABLE SCOPES & PLACEMENTS IN/OUTSIDE FUNCTIONS

GLOBAL
- If variable is declared outside a function then you can use it anywhere in the code.

LOCAL
- If a variable is declared inside a function it can only be used within the function.
- Declare variables with var inside a function to *make it local*, otherwise, it is automatically global
- Using local variables allows you to use the same variable multiple times.
- Use local variables so that it is specific to a function, otherwise, if you use only global variables, you can end up using the same global variables in multiple functions serving different purposes
- Declare it at the beginning of the function for readability/ good practice

IMPORTANT:

- Forgetting to declare local variables can cause problems if you have the same name for another global variable. You may overwrite the value.
- Local variables overshadow global variables

Good Review : HFJS pg 113

Getting Lengths	

String	Ways to check get string length:
	`string.length`
	`string[i] !== undefined`
	
	When string[i] IS NOT equal to undefined, this means it calculates to the end of the string. After the end of the string, the [i] would be undefined.
	
	Ways to check a string is empty:
	`!string.length`
	`string.length === 0`
	`string === ‘’`
	`!string == null`
	`!string`

Array	Ways to check get string length:
	`arr.length`
	`arr[i] !== undefined`
	
	When arr[i] IS NOT equal to undefined, this means it calculates to the end of the arr. After the last [i] in the length, the next would be undefined as there is no value.
	
	Ways to check a string is empty:
	`!arr.length`
	`arr.length === 0`
	`!arr == null`
	`!arr`

Objects	obj[key].length

Mathematical Operators:
When doing *= and setting a variable, set to 1 instead of 0, otherwise, results will be 0.
See BB114 - computeProductOfAllElements

STRINGS

STRINGS	
Definition	

Concatenate Strings	

Accessing Strings	**2 Ways to Access:**
	`string[i]`
	`string.chArt(i)`
	Grabs particular letter of a string at `[i]`
	
	Example:
	`var name = 'Susie'`
	`name[0] = 'S'`
	`name.chArt(1) = 'u'`

Converting Strings to Numbers
 	
Notes
	`var num = '99'`
	
	`parseInt(num,10)`
	works for whole numbers only, otherwise it rounds up if there is a decimal
	`parseFloat(num)`
	works for numbers with decimals
	`~~num`
	doesn't work for decimals
	returns 0 if there any letters
	`num/1`
	`num * 1`
	`+num`
	`Number(num)`
	works for whole numbers only

Suggested Code Review	BB123

ARRAYS

ARRAYS	
Definition	Allows you to store multiple values together. Variables only store one value. Are a special type of object

Syntax	`var arrayName = [`
	`'strings',`
	`true,`
	`99,`
	`var objName = {`
	`name: 'Susie',`
	`age: 29,`
	`city: 'Seattle'},`
	`var stuff = [ 'a', 'b', 'c']`
	`];`
	
	• stores strings, booleans, numbers, objects, other arrays
	• Contained within [ ] brackets
	• separated by commas at the end of each value

Accessing	Use index starting at 0
	Example:
	`Let fruit = ['apple', 'orange', 'banana']`
	`fruit[0] = apple`
	`fruit[0][0] = 'a' of apple`
	
	If array index is too big or small → results in **undefined**
	Example: fruit[4] or fruit[-1] → undefined
	
	Avoid this be checking to see if the index value is not undefined (see Length)

Length	**Getting Length of Array**
	`arrayName.length`
	`arrayName[i] !== undefined`
	Checks to make sure array at a specifc index value is not undefined; if array is sparse
	
	Getting Last Index/Item of Array
	`arrayName[arrayName.length-1];`

Adding to Array	**Manually adding via specific index**
	`fruit[4] = 'pears'`
	
	arrayName.push(new item) method
	`fruit.push('blueberries');`
	
	Adds new value to the end of the array

ARRAY METHODS		
Method	**Action**	**Return**
arr.shift	Removes first element from original array	Removed element
arr.slice	Removes portion of array from beginning/end	New array
arr.splice	Removes / adds elements to an array	Array with deleted elements

Math.floor - rounds a number with decimals up
Math.ceil - rounds a **negative** number with decimals down

Example
math.floor(-4.333) → 5
math.ceil(-4.333) → 4

Math.max cannot pass through an array, only numbers.

In order to use Math.max on an array, use spread operator (…)
Example:
To get the integer length of the elements in fruit, we can create a new array to contain the lengths:
newFruit.push(fruit[i]);

To get the largest number of newFruit (which is the new array containing lengths) using Math.max, we do this:
Example:
Math.max(newFruit); → this does not work because you cannot take in a array in Math.max
Math.max(...newFruit) → this will work because we are able to use spread operator ES6 → it interpolates the values of the arrays out

How to grab the last element of array

Array[array.length-1] → value of last element of a specific array set
Array.length-1 → the value of the last element

OR we can also use array method, pop
Array.pop → will give you the last value

OBJECTS

Objects	
Definition	Collection of properties

Syntax	**Simplified**
	`var obj = {`
	`        key1 : value1,`
	`        key2 : value2,`
	`        key3 : value3`
	`}`
	The key and values of an object are called **properties**.
	Properties, otherwise, known as the [key][value] are separated by commas. Contained within { }.
	Example
	`var chevy = {`
	`        make : 'Chevy',`
	`        year: 1957,`
	`        convertible: false,`
	`        tires: {`
	`                'Winter',`
	`                'All season`
	`                }`
	`        Drive: function () {`
	`                alert.log('Vroom, vroom!');`
	`        }`
	`}`

	• Keys store values. • Values can be: strings, numbers, booleans, objects, functions/methods • Functions/Methods do not have a function name after it, instead it assumes the name of the key. So the function is essentially function drive(); • Functions inside an object are called **methods.**

Accessing	**Example**
	`Var food = {`
	`    type: ['pizza','burgers']`
	`    cost: [1,2,3]`
	`    time: {`
	`        0800: 'Breakfast',`
	`        1200: 'Lunch',`
	`        1800: 'Dinner'`
	`        }`
	`}`
	--- 	
	Objects are accessed by property **keys** not their indexes, like arrays/strings.
	
	To access the 'Lunch' value we do not access it using the indexes.
	WRONG
	`food[2] = 'time'`
	`food[2][1] = 'Lunch'`
	FALSE when accessing an object
	
	To access properties we use dot or bracket notations:
	CORRECT
	Dot Notation → `food.time;`
	Bracket Notation → `food[time];`
	--- 	
	However, you can access object property keys via index **if and only if** you are accessing a key whose value is an array/strings.
	
	`obj[key][i]`
	
	What is `[i]` ?
	• Index of value of the key assuming `i` is a number
	• Has to be a string or an array (not an object)
	• In `obj[key][i]` you are grabbing the object[key] at a specific index of a string or array.
	
	EXAMPLES OF ACCESSING USING INDEX
	`food[time][i]` → **FALSE**
	
	You cannot access the index of `food[time]`, because `time` is a object. There are no indexes in the object, only `key[values]`
	
	To access an object, you would need to specify the key.
	`food[time]['0800']= 'Breakfast'`

	To access an array or sting, you access via indexes. `food[type][ 1 ] = 'pizza'` → TRUE `Obj [key] [i] = 'value at index'` We are accessing the food object, the type key whose value is an array so we are accessing the 1st index. See Bracket vs. Dot Notation Chart below

Change
Add
Delete
	Change property by assigning new value (=)
	`food.cost = [10,11,12]`
	
	Add new property by specifying the new property and giving it a value. Here we add the key dessert and assigned it a value.
	`food.dessert = ['cakes','yogurt','pie']`
	
	Delete property
	`delete food.cost;`
	
	Deletes the entire property, not just the value, cost.
	When deleting a property, it will result in undefined.
	Delete returns true if delete was successful.

Testing Objects for Equality	**Testing Object Equality** ● It doesn't matter if you use == or === ● Only way test for equality between two objects is true is if the **two references** to the same object ○ It *doesn't* matter if the object properties (key[values]) are the exact the same ○ It **needs to point to the same object** ● Better explanation found here.
Length	obj[key].length

BRACKET NOTATION	DOT NOTATION
Good to use if you don't know the key : values	Good to use for defined key : values
Can access numbers/integers, but you need to put the number in strings. `food.time['1200'] → 'Lunch'` → TRUE	Cannot access numbers/integers `food.time.1200 = 'Lunch'` → FALSE

Can access variables that may have spaces, but you need to put it in strings.
`obj = {`
`        first name:'bob'`
`}`
`obj['first name'] = 'Bob'`
→ **TRUE**
	Cannot access variables with spaces.
	There is a space between first / name in the obj.
	`obj.first name = 'Bob'`
	→ **FALSE**

	Used to call the methods in an object.
	`chevy.drive();`
	To access the method, access the key using dot notation, followed it by (). For values, that are not methods:
	`chevy.make;`

	Use `this` in a method to refer to the object whose method was called.

Recursion
A function that calls itself
Why would you want to call yourself?

- Similar to a loop
- Keep in mind memory concerns
- Initial statement ; increment; terminating condition (what ends the loop) you must always have a terminating condition

```
Function countDown(n) {
        Print(n)
        countDown(n-2)
}
```

This would never end because there is no terminating condition, so you'd end up getting

```
        n = 3
        n = 2
        n =1
        n = 0
        n = -1
```

We would revise it as follows:

```
function countDown(n) {
        if (n <= 0)
                Print('Happy New Year')

        else {
                print(n);
                countDown(n-1)
        }
```

Terminating condition if this is not met, then it would call the function countDown again w/ (n-1)

For loops

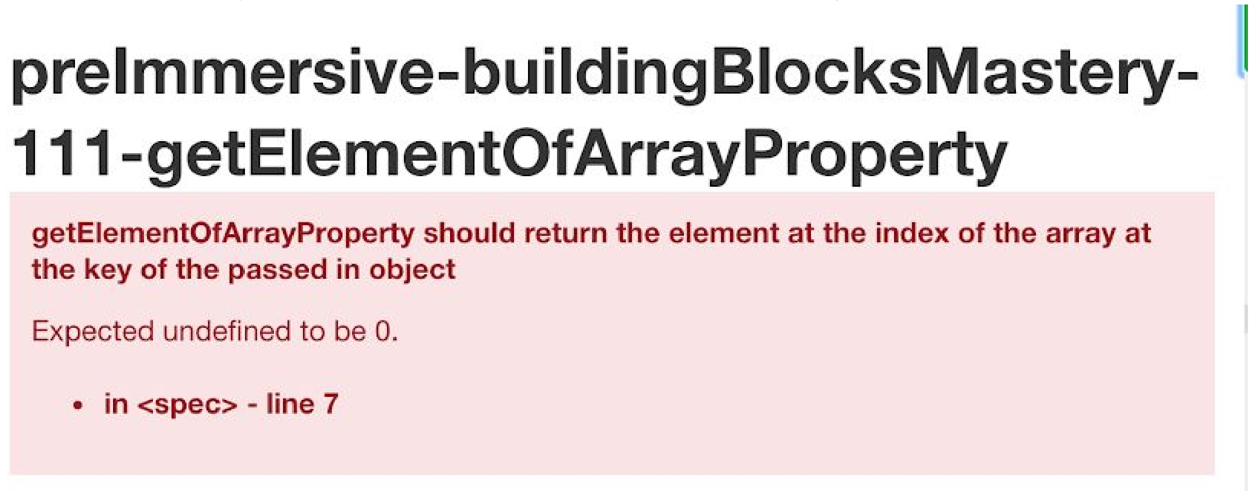
\\ === \

BB107
BB114
BB 117/119
BB 123
BB 124 - concanting arrays - replacing variables values via loops
BB 126 - being aware of converting numbers to strings and back to numbers in order to add it via arithmetic instead of concanting
Remember: strings concant ‘4’+’4’ = ‘44’
Numbers follow arithmetic 4+4 = 8
BB 127

- Using Infinity to set parameter
- Higher order functions
- Array sort / unshift

BB110 - **How come I cannot do obj[key][index] in my logical / conditional statement to verify that the index exist within the obj[key] / array?**

When I put this in my conditional statement, it errors out to the following



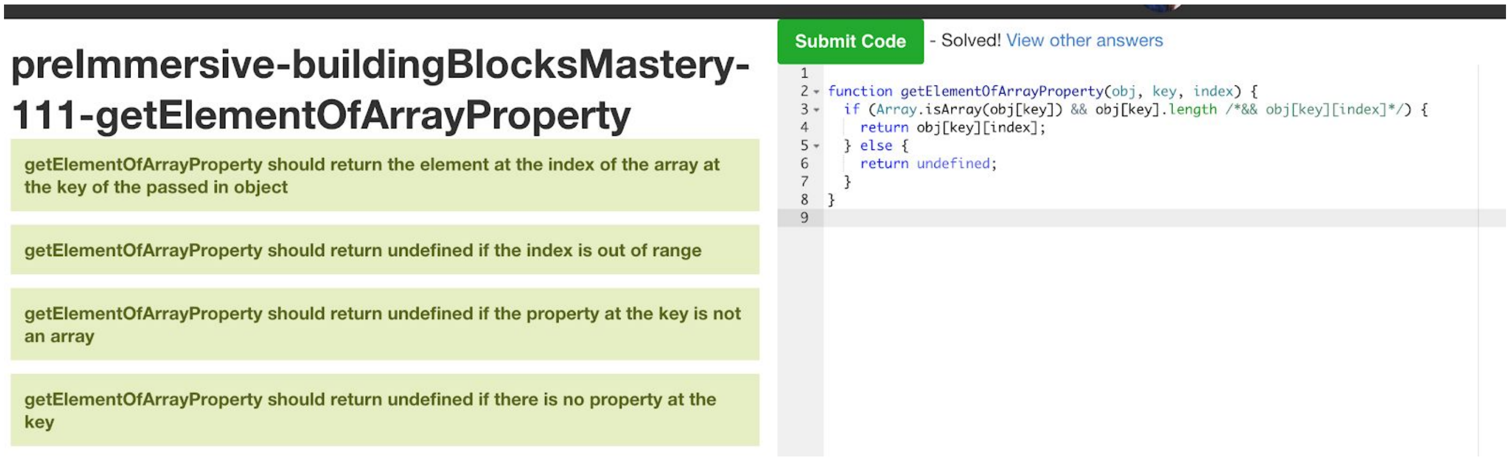
preImmersive-buildingBlocksMastery-111-getElementOfArrayProperty

getElementOfArrayProperty should return the element at the index of the array at the key of the passed in object

Expected undefined to be 0.

- in <spec> - line 7

When commenting out the obj[key][index], it passes all requirements?



preImmersive-buildingBlocksMastery-111-getElementOfArrayProperty

getElementOfArrayProperty should return the element at the index of the array at the key of the passed in object

getElementOfArrayProperty should return undefined if the index is out of range

getElementOfArrayProperty should return undefined if the property at the key is not an array

getElementOfArrayProperty should return undefined if there is no property at the key

```

function getElementOfArrayProperty(obj, key, index) {
  if (Array.isArray(obj[key]) && obj[key].length /*&& obj[key][index]*/) {
    return obj[key][index];
  } else {
    return undefined;
  }
}

```

ANSWER: The way the code would have worked is that it would have passed the conditional statements assuming true, then when it got to the `return obj[key][index]` portion it would say *oh this doesn't exist*... so then it goes to `else { return undefined; }`