Contents

[TOC]

Javascript Basics

Primitives and the Console

- Basic Building Blocks in Javascript are 1. Number 2. String 3. Boolean 4. Null 5. Undefined
- Other two technical types (Way less commonly used.) 1. Symbols 2. Big Integers

Numbers

- Comments
- Using doubleslash
- //This is a comment
- **Numbers**
  - Javascript has one number type. Some other languages have more than one.
  - Positive Numbers
  - Negative Numbers
  - Whole Numbers(Integers)
  - Decimal Numbers
- **Math operations**
  - Addition.
  - Subtraction.
  - Division.

  - Multiplication.
  - Exponentiation.
  - Modulo.

- Not a Number(NaN)

  - NaN is a numeric value that represents something that is not a number.
  - **NaN in Javascript is considered as a number.**
  - The reason NaN in Javascript is considered as a number is because it is technically a numeric data type whose value cannot be represented using actual numbers.
  - Any operation along with NaN is NaN
  - Example : 0/0 - NaN = NaN,
    * 1 + NaN - NaN = NaN
  - There are five different types of operations that return NaN:
    1. Number cannot be parsed (e.g. parseInt("blabla") or Number(undefined))
    2. Math operation where the result is not a real number (e.g. Math.sqrt(-1))
    3. Operand of an argument is NaN (e.g. 7 ** NaN)
    4. Indeterminate form (e.g. 0 * Infinity, or undefined + undefined)
    5. Any operation that involves a string and is not an addition operation (e.g. "foo" / 3)

Operators in Javascript. - Operators in Javascript - In short. 1. Arithmetic Operators (+,-,*,**,/,%,++,–) 2. Assignment Operators (=,+=,-=,*=,/=,* *=,%=) 3. Comparison Operators (==, ===, !=, !==, >, <, >=, <=, ?) - Note that '===' and '!==' don't only check the value but they also look for the type of the variable. 4. Logical Operators (&&, ||, !) 5. Type Operators (typeof, instanceof) 6. Bitwise Operators (&, |, ~, ^, «, », »>) - Escape Sequences 1. (') : Single quote 2. (") : Double quote 3. (\) : Backslash 4. (\n) : newline 5. (\r) : carriage return 6. (\t) : tab 7. (\b) : word boundary 8. (\f) : form feed

Variables

- Variables are like labels for values
- We can store a value and give it a name so that we can:

  - Refer back to it later
  - Use that value to do… stuff
  - Or change it later one

- Basic Syntax

  - let someName = value;

- Example

  - let year = 1985;
  - console.log(year)

Variable types

- let
- const
  - const works just like let, except you cannot change the value.
  - const variable does not allow to change the value.
  - We can use it to change the things we know won't change.
  - Example : const pi = 3.14159;
- var
  - Before let and const, var was the only way of declaring variables. These days, there isn't really a reason to use it.

Booleans

- Booleans are used to store True or False values.
- Unlike Python , they are lowercase in Javascript.
- You can change a number variable to boolean or any other type in javascript.

Variable naming and conventions

- In Javascript, identifiers are **case sensitive**, it can contain Unicode letters,$,_,digits but may not start with a digit.
- Give variables camelCase names.
- Give proper variable names (which can be understood.)

Javascript Strings and More

Strings

- Strings are another primitive type in Javascript.
- **Strings in Javascript are immutable**
- They represent text, and must be wrapped in quotes(single, double or triple).
- Strings can also be empty : “”
- Strings can be added to each other i.e concatenated
- Syntax:

  - `"String 1" + "String 2"`
  - `result+="String1"` (When you want to modify the string result by concatenating it with other string)

- Similarly, we can also concatenate a string and a number.
- The number will be converted to string when concatenating it with a string.
- Example - 1 + “hi” = 1hi(which is a string)
- Also, we can overwrite a string to change its value.

Indices and length

- Index
- Strings are indexed i.e each character has a corresponding index (a positional number).
- Index starts from 0.
- We can access individual characters based on their index.
- **But we cannot change the individual characters with the help of their index as strings are immutable in Javascript.**
- Length
- Syntax : `var.length`
- **length starts counting from 1 unlike index which starts counting from 0.**
- **It is not a method but rather a property. So we do not add parentheses () unlike other methods.**

String Methods

- String Methods
- Methods are built-in actions we can perform with individual strings.
- They help us do things like:
  - Searching within a string
  - Replacing part of a string
  - Changing the casing of a string
- Syntax : `thing.method()`
- Methods

  - We will identify methods based on **destructive or non destructive**. Destructive methods change the value of the original string without overwriting when the method is applied on it whereas non destructive method preserves the original string.
  - Types of Methods
    * toUpperCase() - Changes all the characters of the string to Upper case (Non destructive method)
    * toLowerCase() - Changes all the characters of the string to Lower casw (Non destructive method)
    * trim() - Removes any whitespace at the beginning of the string or at the end (Non destructive method)

- We can also chain methods together. Example: JS `string.trim().toUpperCase();`

String Template Literals

- Template literals are strings that allow embedded expressions, which will be evaluated and then turned into a resulting string.
- Expressions can be embedded using ${expression} where we can use variable or an expression.
- ‘String is enclosed with the use of backticks.‘

Undefined and Null

- Null

  1. Intentional absence of any value is given as null.
  2. Must be assigned.

- Undefined

  1. Variables that do not have an assigned value are undefined.

Random number and Math Object

- Math Object

  1. Contains properties and methods for mathematical constants and functions.

Decision Making

Comparison Operators

- Comparisons

1. '>' ….. greater than
2. '<' ….. less than
3. '>' ….. greater than or equal to
4. '<=' ….. less than or equal to
5. '==' ….. equality
6. '!=' ….. not equal
7. '===' ….. strict equality
8. '!==' ….. strict inequality

- Comparisons are usually used for numbers however we can also use it for strings where we compare the unicode values of the strings.
- All of the comparison operators return a boolean true or false value.
- Example

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

- The better way to write this is: `JS function isEqual(a,b) { return a === b; }`
- This helps only in case of booleans as the comparison operators return true or false

Equality

```
a = 14
b = "14"

console.log(a == b) //true
console.log(7 == "7") //true
console.log(0 == "") //true
console.log(0 == false) //true
console.log(null == undefined) //true
console.log("b" == "c") //false
console.log(true == false) //false

//In Javascript
```

```
//Number("") == 0
//Number("923") = 923
//Number("923 123") = NaN
// Number("jskfdjd") = NaN
```

Console, Alert Prompt

- console.log()
  - prints arguments to the console.
- alert()
  - alerts a message in the browser or DOM
- prompt()
  - prompts the user asking for value in the browser window.

If Statement

- Conditional statements is making decisions with our code.
- if statement only runs code inside the curly braces if the condition is true.
- If the condition is false, nothing happens or either true elif or else condition is executed.
- Syntax: `JS if(condition){ code } else if(condition){ code } else{ code }`
- Check code too for better understanding.

Truthy and Falsy

- All JS Values have an inherent truthyness or falsyness around them.
- **Falsy Values**
  1. false
  2. 0
  3. ""(empty string)
  4. null
  5. undefined
  6. NaN
- Everything else is truthy

Logical Expressions

- Logical operators are the ones that allow us to combine different expressions.
- So we can combine more than one piece of logic together to form one larger piece of logic.
- Logical operators

    1. AND (&&): Both sides must be true for entire thing to be true.
    2. OR (||): If one side is true, the entire thing is true.
    3. NOT (!) : !expression returns true if expression is false and false if true.

- **AND has precedence over OR. You can use parentheses to filter the precendence**.

Switch Statement

- The switch statement is another control-flow statement that can replace multiple if statements.
- Syntax

```
switch(variable or number){
    case 1:
        //code;
        break;
    case 2:
        //code;
        break;
    case 3:
        //code;
        break;
    case n:
        //code;
        break;
    default:
        //code;
}
```

- Working
    - So switch detects the number or variable passed into it.
    - It keeps on executing code ahead of it until the end of the loop or break .
    - So if there are 7 cases without break and the parameter if 5, it will execute 5,6 and 7th cases.
    - That's why break is necessary.
- Example JS `function sequentialSizes(val) { var answer = ""; switch(val){ case 1: case`

```
2:            case 3:            answer = "Low";            break;
case 4:            case 5:            case 6:            answer
= "Mid";            break;            case 7:            case 8:
case 9:            answer = "High";            break;        }
return answer;     }     sequentialSizes(1);
```

- In the above example, answer will be Low for cases 1,2 and 3, Mid for cases 4,5 and 6, High for cases 7,8 and 9.

Javascript Arrays

Introduction

- Ordered collection of values
    1. List of comments on IG Post.
    2. Collection of levels in a game.
    3. Songs in a playlist.
- Creating Arrays.
    1. To make an empty array `let students = [];`
    2. An array of strings `let colors = ['red','orange','yellow'];`
    3. An array of numbers `let lottoNums = [19,22,56,12,51];`
    4. A mixed array `let stuff = [true, 68, 'cat', null, undefined]`
    5. **Javascript can also create a mixed array unlike C++.**
- Array Random access.
    1. Just like strings, arrays are also indexed. Each element has a corresponding index where the counting starts at 0.
    2. Example `colors[0] //red colors.length //3`
    3. If index is not present in the array, you get undefined.
    4. We can also chain indexes `colors[0][0] //r`

Modifying Arrays

- **Arrays can be modified unlike strings which cannot be modified but needs to be overwritten.**
- Example JS `let colors = ['rad','orange','blue','white','black']; colors[0]= "red";`

Array Methods

- Resources

    1. MDN Arrays - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
    2. Slice, Splice, Split - https://www.freecodecamp.org/news/lets-clear-up-the-confusion-around-the-slice-splice-split-methods-in-javascript-8ba3266c29ae/

- **Array Methods**

    1. Push - add to end (destructive method)
    2. Pop - remove from end (destructive method)
    3. Shift - remove from start (destructive method)
    4. Unshift - add to start (destructive method)
    5. concat - merge arrays(non-destructive for original arrays)
    6. includes - look for a value (false if not present)
    7. indexOf - returns the index of the array (-1 if not present and only returns first occurrence)
    8. join - creates a string from an array
    9. reverse - reverses an array (destructive method)
    10. slice - copies a portion of an array
        1. takes two optional parameters
        2. same as strings, index starts from 0 and goes till n-1.
        3. For negative index it will start from end of the index and assume that the index will start from 1 from the end.
    11. splice - removes / replaces elements
        1. Syntax array.(start_index, delete_count, items)
    12. sort - sorts an array
        1. Compares everything to string and then compares their UTF values

Reference and Equality Types

- `[1] === [1] or [1]==[1] or ['']==['']`

    1. This will give false as in arrays we are not comparing the content.
    2. Javascript doesn't care what's inside, atleast with arrays.
    3. **What it compares instead are the references in memory.**
    4. Let luckyNum = 87; Certain amount of space is allocated in memory for this number.
    5. For arrays that's not the case, you can have many many elements in an array where it stores lots of space.
    6. luckyNums = [1,2,3]

7. Instead of associating luckyNums with all elements in luckyNums array, what Javascript does is it stores a reference which corresponds to array.
8. So if you make another [1,2,3], it is actually a different array. That is a new array in memory is created.
9. **Refer the reference code as well.**
10. Check the below code

```
let nums = [1,2,3]
numsCopy = nums
console.log(nums)
console.log(numsCopy)
nums.push(4)
console.log(nums)
console.log(numsCopy)
```

11. In the above code, numsCopy doesn't create a different copy of nums, however, it references to the same array as nums. So any changes made to numsCopy will reflect to nums and vice-versa.

Arrays and Constant

- For strings we cannot change the value of a const
- For arrays, the shell remains the same but the content can change
- **Updating the content doesn't change the address of the array, the reference.**
- **Example - As long as egg carton remains change you can put different eggs in it and change.**
- As soon as you try to change it to new reference, you get an error.
- **In short, the difference between strings and arrays is that the strings are immutable and the arrays are mutable. However, in case of const, we cannot re-assign values to both of them.**

Multidimensional Arrays

- Also known as Nested Arrays.
- We can store arrays inside other arrays.
- Refer code.

Javascript Objects

Object Literals

- Objects are collections of properties.

- Properties are a key-value pair.
- Rather than accessing data using an index, we use custom keys.
- It consists of property key + value
- It is similar to the dictionaries in python.
- Example: JS `const fitBitData = { totalSteps : 308727, totalMiles : 211.7 avgCalorieBurn : 5755, workoutsThisWeek : '5 of 7' avgGoodSleep : '2:13' }`
- All valid keys are converted to strings in objects.(Except for symbols - which is uncommon)

Accessing Data Objects

- Let us start with an example

```
person = {
    firstName : "Yash",
    lastName : "Thakkar"
}
```

- In order to access firstName

```
person["firstName"]
person.firstName
```

- []
    1. For a key which is a string, it expects it to be a variable name in [], therefore you need to wrap it up with double quotes.
    2. However, you can let numeric, boolean keys in double quotes as well as without it
- dot access
    1. Whereas in dot method, you shouldn't wrap it in double quotes.
    2. Dot method doesn't work for numbers.
    3. It doesn't support variables as well

Object Methods

```
const counterStrikeRating = {
    Morris : 85,
    Cooper : 80,
    Ben : 88,
    Sam : 95,
    Maverick : 99,
```

```
    Yash : 100,
    Ace : 96,
    Tex : 96,
    Steel : 97
}
```

1. Object.keys(counterStrikeRating)
2. Object.values(counterStrikeRating)
3. Object.entries(counterStrikeRating)
4. counterStrikeRating.hasOwnProperty(propertyname)
5. Example `counterStrikeRating.hasOwnProperty("Morris");`

Modifying Objects (Imp)

- Check out the `03_imp_modifying_objects.js` code.

Array Objects

- Example JS `const student = { firstName : 'Yash', lastName : 'Thakkar', strengths : ['Music', 'Programming'], exams : { midterms : 95, final : 100 } }`
- Example JS `const shoppingCart = [ { product : "POCO C3", price : 7000, quatity: 10 }, { product : "POCO M2", price : 9000, quatity: 5 }, { product : "Asus Max Pro M3", price : 9000, quantity: 1 } ]`

Loops

For Loop

- Loops allow us to repeat code
- Sum all numbers in an array.
- There are multiple types:
    1. for loop
    2. while loop
    3. for…of loop

4. for…in loop

- For Loop Syntax JS `for( [initialExpression]; [condition]; [incrementExpression] )`
- Example JS `for (let i = 1; i<=10; i++){ console.log(i) }`
- The loop keeps on executing until the condition is false
- Hence

1. The first part is the initialization part
2. The second part is the condition part, if it is true, the loop will get executed, else it will not
3. The third part is the incrementing part, so that the condition can be false and the loop comes to an end.

Infinite Loops

- Infinite loops should be avoidable.
- Loops which keep on going as there is no false condition are known as infinite loops and they keep on running.
- They take up computer memory.

Looping Arrays

- To loop over an array, start at index 0 and continue looping to until last index (length-1)
- Example JS `const animals = ['lions','tigers','bears']; for(let i = 0;i<animals.length;i++){ console.log(i,animals[i]); }`

Nested Loops

- Example JS `let str = 'LOL'; for (let i = 0 ;i<=4; i++){ console.log("Outer",i) for (let j = 0; j<str.length;j++){ console.log("Inner:",str[j]) } }`
- For every single iteration of the outer loop, let's say it runs five times, the inner loop is going to have it's own full cycle
- while i is 1, we have an entire nested loop which runs for j starting at 0 upto 2. So we end up with 3 iterations of j all when we are at 1st iteration of i, then we move up where i is 2 and the whole process repeats

- This is how it works `0 0 1 2 1 0 1 2 2 0 1 2 3 0 1 2 4 0 1 2`

While Loop

- while loops continue running as long as the test condition is true.
- Example let num = 0 while (num < 10){ console.log(num); num++; }
- break keyword

    1. break keyword breaks out of the while loop.
    2. It is especially useful in case when we want to loop a statement again and again except for a certain condition.
    3. We can use break keyword for that particular condition.

For of Loop

- Syntax `JS for (variable of iterable){ statements; }`
- for of loop iterates through the whole string and array

Iterating Objects

- for in loop is used to iterate over the objects.
- Its Syntax is `JS for (variable in iterable){ statements; }`
- using for in you can get the key of the key-value pair.
- in order to get the value, you can write `console.log(iterable[variable])`
- Other special methods for objects

    1. Object.keys(Object_variable) //Returns array of keys
    2. Object.values(Object_variable) //Returns array of values
    3. Object.entries(Object_variable) //To get nested array of key-value pairs

To-Do Exercise (Refer Code)

- This is a basic to do app.
- We will keep asking for choice until the user decides to quit.
- We have created an empty array in the beginning.
- If user enters new, and enters a todo. The todo is pushed into the array.
- If the user wants to view his todo, then we will parse through the todo array.

- i+1 is used to show first todo index as 1.
- if choice is delete, we will ask user for the index of todo to delete and parse it to int as prompt accepts input as string.
- Then we will check if the input entered by the user in prompt is actually a number or not.
- If it is NaN(that is isNaN is true), then the first condition will get false and the else loop will be executed and valid index will be asked.
- If it is not NaN, then the first condition will get true and the if loop will be executed.
- In the if loop, we will make use of splice to remove the element.
- remove-1 is used as the index is shown incremented as discussed in point 6.
- Then one element is removed at that index and the deleted index and the name of the deleted todo is shown on window.

Functions

Introduction

- Functions are reusable procedures
- Functions allow us to write reusable, modular code.
- We define a “chunk” of code that we can execute at a later point.
- We use them all the time.
- It is a 2 step process

    1. We need to define the function
    2. We need to run the function

- Defining a function `JS function funcName(){ //do something }`
- Always define the function before you use them.
- Also, technically functions are objects behind the scenes.

Arguments

- We can also write functions that accept inputs, called arguments !
- We have seen this before

    1. “hello”.indexOf(‘h’) //We pass ‘h’ as argument here
    2. nums.push(5,6,7)

- Example `JS function greet(person){ //Parameter console.log(`Hi, ${person} !`); } greet("Yash") //Arguments`
- If an expected argument is not passed to the function, it’s gonna have a value of undefined.

- And if an extra argument is passed to the function, it accepts the first argument and discards the second.
- This happens only in Javascript and not in other languages.
- **Functions with Multiple Arguments**

    1. We can pass multiple arguments in a function.
    2. The arguments are matched to the parameters with respect to the order.. That is first parameter will hold the value of first argument, second of second argument and so on.

Return Statement

- return keyword is used to capture the return value of a function to a variable.
- The return statement ends function execution AND specifies the value to be returned by that function.
- That is statements after the return keyword in a function are not executed.

Leveling up our Functions

Scope

- Scope - Variable “visibility”
- The location where a variable is defined dictates where we have access to that variable.
- Variables we define inside a function are scoped to that function, therefore if we use a function variable outside a function, then we get an error.
- Example `JS let msg="I am on water!" function helpMe(){ let msg = "I am on fire!"; console.log(msg); } console.log(msg); // I am on water!` Here the msg is scoped to the helpMe function.
- If there is a variable defined by the same name in the function, in the function scope, then the closer reference will be used. That is in a function for console.log closer, if there is a variable with the same name as outside the function, the function’s variable will be used, else the outer variable.
- **In case of let, the variables defined inside the function are not usable outside it.**
- Example `JS let bird = "mandarin duck" function birdWatch(){ let bird = "goldenpheasant"; bird; //goldenpheasant } bird; //mandarin duck`
- Although, if you declare a variable outside the function, change it in function, call the function and then use it. Then you can get the changed value inside the function PROVIDED it is not declared inside the function.

- This also means that a variable declared outside the function is available to the function too.
- Variables which are defined outside of a function block have Global scope. This means they can be seen everywhere in your Javascript code.
- Also in order to use it, you don't need use global keyword in a function unlike python where if you want to modify the same global variable, you need to specify it as global.

Block Scope

- Variables defined in a block are just scoped to the block.
- Blocks refers to anytime we see curly braces except for a function.
- Example : JS `let radius = 8; if (radius>0){ const PI = 3.14; let circ = 2*PI*radius; } console.log(radius); //8 console.log(PI); //Not Defined console.log(circ); //Not Defined`
- Here, PI and circ are scoped to the block
- Same as functions, if the variables are declared outside the block and later modified in the block and then used outside, the output will be changed value PROVIDED it is not declared inside the scope.
- This also means that a variable declared outside the block is available to the block too.

Declarations

- Using var keyword, variables are scoped to functions but they are not scoped to blocks.
- That is, when var keyword is used inside a function, the variable is not available outside the function, however, in case of blocks when var is used, the variable is available outside the block as well.
- let and const are block and function scoped.
- When you have local and global variables with the same name , the local variable takes a precedence over the global variable.
- It is usually recommended to avoid the use of var keyword.

Lexical Scope

- An inner function nested inside of some parent function has access to the scope or the variables in the scope of that outer function.
- Example "'JS function outer(){ let hero = "Rune King Thor";

```
function inner(){
    lt cryForHelp = `${hero}, please save me !`
```

```
        console.log(cryForHelp);
    }
    inner();

}"'
```

- If something exists in outer(), we have access to it in inner() but this doesn't work the either way.

Scopes Final

- So, there are scopes such as function scope, block scope and lexical scope.
- Function scope
  - Variables declared inside the function are scoped to the function.
  - If two variables with same name are declared, then the one having closer reference will be used.
  - We can declare a variable, **change** the variable in function, then call the function and then can use it in program outside of that particular function.
  - Variables declared outside the function are available to the function but not vice-versa
- Block Scope
  - Just as functions, variables declared in the block are just scoped to the block.
  - Blocks refer to anytime we see curly braces except for a function such as if, for, while, etc.
  - Rest it has all properties same as functions except for one thing and that is var keyword.
  - var keyword is not block-scoped and is just function scoped or globally-scoped.
  - That is variables declared using var keyword in the blocks, is available to the global scope as well.
  - However, it is recommended to avoid use of var keyword and just use let and const and hence, block and function scoped are usually same for variables declared with these two declarations.
- Lexical Scope
  - An inner function nested inside of an outer parent function has access to the variables in the scope of that outer function.
  - However, it is not the same case vice-versa.

Function Expressions

- There is a different way of defining a function and it actually involves storing the function in a variable.

- Example JS `const square = function(num){ return num*num; } square(7);`
- The above is a function expression, the right side after the variable add creates a function.
- This function is stored in a variable that is square.
- It is same as storing variable, object, array or a number. It is the same exact concept.
- It is used in same way as any other function.
- Here square is not the name of the function but name of the variable. That is we are storing a function with no name inside of a variable.
- Whereas in normal cases, we make a function with its own name. Both behave the exact same way.
- The first case is just like storing a value in variable. Example
  - const PI = 3.14
  - 3.14 does not have a name but the variable(or container) does have a name.
- Functions are values in Javascript, we can store them, we can pass them around just like number, array, string. Javascript considers functions just like any other value
- If you call the square as square, you'll be returned the entire function and to use it you have to call it by doing square() just like normal functions.

Higher Order Functions (Important)

- Higher order functions are functions that work with other functions so that they operate on/with other functions.
- They can
  1. Accept other functions as arguments.
  2. Return a function.
- Function as arguments example: "'JS function callTwice(func){ func(); func(); }

  function laugh(){ console.log("HAHAHAHAHAHA") } callTwice(laugh) "'
- Here, a mistake not to be made is passing laugh as laugh().
- Because what this will do is it will execute the laugh and output the result and then pass "HAHAHA…." as an argument.
- But this is not what we want to do, what we want to do is pass through the value of the function i.e laugh so that inside of callTwice it can be executed.

Returning Functions (Important)

- Refer the Returning functions code.
- If you call the makeMysteryfunc(), it will return the entire function.
- You need to capture the entire return value.
- Most important thing to remember is, if you are calling the main function, you need to save the function in a variable. And then call the variable with the inner function's parameters.

Methods

- We can add functions as properties on objects.
- We call them methods.
- Example to create a method JS `const math = { multiply : function(x,y){ return x*y; }, divide : function(x,y){ return x/y; }, square : function(x){ return x*x; } };`
- Some built-in methods are toUpperCase(), push(), indexOf()
- Shorthand method to use Methods in Javascript JS `const math = { blah : 'Hi !', add(x,y){ return x+y; }, multiply(x,y){ return x*y; } } math.add(50,60) //110`

This

- Use the keyword this to access other properties of the same object.
- Example JS `const person = { first : 'Yash', last : 'Thakkar', fullName(){ return `$(this.first) $(this.last)` } } person.fullName() //Yash Thakkar person.last = "" person.fullName() //Yash`
- The value of 'this' can change and it depends on the invocation context of the function it is used in.
- That is it depends on how we call the function.
- Usually when we use a method, like person.fullName(), the object before the dot(.) is invoked. So the value for the this.first would be key first of object person.
- However when we do something like, JS `const fullName2 = person.fullName; fullName2();`
- The fullName2() does not refer to the person object.
- The fullName2() references to the **window object which is the by default value as there is not dot(.) operator** before the fullName2() variable and hence it doesn't refer to any object.

- When you create any function, it is added to the window although you never wrote the window anywhere.

Try-Catch

- Try catch are two statements in Javascript that go together.
- They have to do with errors and exceptions in Javascript.
- Specifically, they have to do with catching errors and preventing them from breaking or stopping the execution of our code.
- We can't have just try we also need to have catch which is a block of code that will run if there was an exception or error generated inside of the try block.
- Generally, code after an error is not executed and the execution of the code stops at the point, the error is found.
- However, using try/catch block we can avoid that.
- Syntax: `JS try{ code; } except ([e]){ code in case of error; }`
- Here e is optional, and e is the error which usually is printed out in case of an error. We can also print the error if required.

Arrays and Callback Methods

For each

- Reference - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEac
- What for_each does is present in the name itself, that is the for_each calls the function for each element in the array.
- Accepts a callback function. Calls the function once per element in the array.
- It allows us to run some function i.e run some code once per item in some array.
- So whatever function we pass in, that function will be called once per item where each item will be passed in to the function automatically.
- **The parameter in the function represents each element of the array**.
- For example (refer the code)
- In the code, the movie is the parameter and it represents each element in the array.
- Difference between for_each and map -
- https://codeburst.io/javascript-map-vs-foreach-f38111822c0f
- In map, and filter, we cannot console.log() but rather we return the array to a new variable whereas in forEach we can console.log() or either mutate the array.

Map Method

- Reference - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
- Map method creates a new array with the results of calling a callback on every element in the array.
- Example JS `const texts = ['rofl', 'lol', 'omg', 'ttyl']; const caps = texts.map(function(t){ //The function parameter t represents each element of texts return t.toUpperCase(); }) texts; //["rofl","lol","omg","ttyl"] caps; ["ROFL", "LOL", "OMG", "TTYL"]`

Intro to Arrow Functions

- Newer syntax for defining functions.
- Syntactically compact alternative to a regular function expression.
- Example "'JS const square = (x) =>{ return x*x; } This can also be written as(However this works for functions with just one parameter) const square = x =>{ return x*x; }

  const sum = (x,y) => { return x + y; } "'

Implicit Arrow Function

`Implicit Return - All these functions do the same thing.`

1. const isEven = function(num){ //regular function expression return num%2==0 }
2. const isEven = (num) => { //arrow function with parens around param return num % 2 == 0 }
3. const isEven = num => { //no parens around params return num % 2 == 0 }
4. const isEven = num => (//implicit return num % 2 == 0);
5. const isEven = num => num%2 == 0; //one-linear implicit return
6. Implicit returns just works when there's a single clear value to be evaluated.
7. It doesn't work in the cases like this JS `const rollDie = () => { let greet = "Hello" return Math.floor(Math.random()*6+1) }`

Sorting out Functions

1. Functions //Normal Functions function funcName([parameters]){ //do something. }
2. Function Expressions //Another way of writing Functions variable = function([parameters]){ //do something }

3. Higher Order Functions //Functions that work with other functions and can accept other functions as arguments and return a function function funcName(funcParameter){ funcParameter(); }
4. Returning functions // Can return a function but need to capture the value of the outer function and then call the inner function function funcName([parameters]){ return function(){ //Do something } }
5. Methods //Can add functions as properties on objects. variable objName = { variable key : value, variable key : function([parameters]){ code; }, funcName([parameters]){ //code } } objName.variable //in case of a value variable objName.variable([parameters]) //in case of a function variable
6. Arrow functions //Syntactically compact alternative to regular function expressions. variable = ([parameters]) =>{ //code return expression } //You can also remove the parentheses in arrow functions in case of a single parameter
7. Implicit return //Just works when single values are to be evaluated or returned variable = (parameters) => (code //No need to write return statement) variable = parameter => expression //One linear implicit return

- Different methods

1. forEach //Allows us to run some function once per item in an array. list.forEach(function([parameter]){ //Here the parameter is used to denote the elements in an array //code }) list.forEach(function)
2. map //Allows us to create a new array with the results of calling a call back on every element on the array. variable = list.map(function)

setTimeout, setInterval

- **Introduction**

1. These are the two functions that expects you to pass a callback function in but they are not array methods.They have nothing to do with arrays.
2. They are used for scheduling execution.
3. In other programming languages, it is known as sleep, pause where we pause execution for some parameter time however in Javascript it is not similar to pause(3000) or sleep(3000)
4. Reference for clearTimeout and clearInterval - https://www.geeksforgeeks.org/javascript-cleartimeout-clearinterval-method/

- **setTimeout, setInterval, clearInterval**

1. setTimeout This function executes the function after set interval of time. setTimeout(TimerHandler,timeout?:number) Here, the TimerHandler is a function basically and

timeout is how long something should take or how long delay should be. 2.setInterval This function repeats the function after set interval of time. setInterval(TImerHandler, timeout)

2. clearInterval Everytime we call setInterval it gives an id or value and we need to save the return value of setInterval, we can have whole bunch of different setIntervals and we can specify which one we want to stop by using this ID. id = setInterval(TimerHandler, timeout) clearInterval(id)

Filter Method

- Creates a new array with all elements that pass the test implemented by the provided function.
- Example JS `const nums = [9,8,7,6,5,4,3,2,1] const odds = nums.filter(n => { return n%2 == 1; //our callback returns true or false //if it returns true, n is added to filtered array }) //[9,7,5,3,1]`
- Example 2 JS `const smallNums = nums.filter(n => n<5); [4,3,2,1]`
- So in short if the function passed to the filter returns true for a given element, filter will return that particular element.

Some and Every

- Both of them are very similar and they are boolean methods meaning they return true or false

1. Sum method
    1. tests whether some elements in the array pass the provided function. It returns a Boolean value.
2. Every method
    1. tests whether all elements in the array pass the provided function. It returns a Boolean value.
3. Example “‘JS const words = [“dog”, “dig”, “log”, “bag”, “wag”];
    1. Every words.every(word => { return word.length == 3; })//true
       Some words.some(word => { return word.length == 3; })//true
    2. Every words.every(word => word[0] === “d”) //false
       Some words.some(word => word[0] === “d”) //true
    3. Every words.every(w =>{ let last_letter = w[w.length-1] return last_letter === “g”; }) //true
       Some words.every(w =>{ let last_letter = w[w.length-1] return last_letter === “g”; }) //true
       “‘

Reduce

- Executes a reducer function on each element of the array, resulting in a single value.
- Example `JS [3,5,7,9,11].reduce((accumulator, currentvalue) =>{ return accumulator + currentvalue; })`
- Accumulator will be a thing, which we are reducing down to. Accumulator variable in above case will hold the sum, current value represents each individual element of the array.
- Reduce in javascript works in the same way as that of Python, that is it does sequential computation.
- Accumulator will be initially the first element of the array, which in the above case, is 3 and then gets changed according to the operation with the current value and currentvalue will get swapped after each callback and in the above case they are 5,7,9 and 11.

This in Arrow Functions (Refer Code)

- **this keyword in arrow functions**
- 'this' keyword in arrow functions behaves much differently in arrow functions than in normal functions.
- Inside of an arrow function, the keyword this is just going to refer to the scope that it was created in.
- So in this case, the keyword this refers to the window object just similar to the previous section where we pointed an object function to a new variable.
- **Understanding the arrow_this.js code**
- person is the object
- thisPrinter is a NORMAL function which returns person object on printing this.
- fullName function is also a NORMAL function which also refers to the person object.
- Now, fullName2() is an ARROW function, and it refers to the window object. Therefore, this.firstName and this.lastName will print as undefined.
- In shoutName, the NORMAL function shoutName actually refers to the person object BUT the NORMAL setTimeout function inside the shoutname function refers to window object.
- Whereas in shoutName2, the setTimeout ARROW function has the same value of this as that of shoutName2 function and hence it will refer to the person object.
- Therefore in short,
- A nested NORMAL function or a NORMAL function in a NORMAL function does not refer to the same 'this' keyword that the parent function is referring and hence it refers to window object.

//code line 37 and here for this.firstName it will print it's parent function firstName that is hello, in case of firstName not defined in parent function, it will print undefined.

- A nested arrow function or an arrow function in a NORMAL function refers to the same 'this' keyword that the parent function is referring. //code line 45
- Arrow functions by themselves in an object does not refer 'this' to the object and hence it refers to the window object.
- Therefore normal functions must be used in an object in order to use the this keyword however if you want to use 'this' in a nested function, you may use arrow functions.