ADVANCED JAVASCRIPT

TOPICS

TOPICS

- ➔ String
- ➔ Template Literals
- ➔ Comments
- ➔ Scopes
- ➔ Hoisting
- ➔ Let Var Const
- ➔ DOM
- ➔ Events

TOPICS

- Object
- Spread Operator
- Rest parameter
- Closures
- Constructor
- Classes
- Inheritance
- Destructuring

TOPICS

- ➔ Storage
- ➔ Conditions Operator
- ➔ Arrow Functions
- ➔ For of loop
- ➔ For in loop
- ➔ Callback
- ➔ Async Js
- ➔ Generator Functions
- ➔ Iterable or Iteration
- ➔ Modules

STRING

String

- Strings in JavaScript are a sequential arrangement of Unicode characters, enclosed inside “ ” or ‘ ’.
- A string is a sequence of characters.
- It can be also defined as a string literal, which is enclosed in single or double quotes:
- Both are valid ways to define a string.
- The string is one of JavaScript’s **Primitive Data Types**.

String

What is primitive Data Types?

A primitive (primitive value, primitive data type) is data that is not an object and has no methods.

Primitive Values:

Are values that has a dedicated memory address values on the **Stack**

Memory:

They are values and doesn’t have properties.

String

- Every programming languages has its own way to manage primitive values. Commonly primitive values are more special in terms of storage. It has a fixed address allocated on the stack memory.

- List of JavaScript primitive value types are ***string, Number, Boolean, undefined, null,*** symbol (new in ECMAScript 2015)***.*** **But they have also have a Primitive methods that they can use.**

String

- In JavaScript, a valid string can be:
- Three ways to define string.
  - Single quote
  - Double quote
  - Backtics (Template literals) will cover in next topic

String

Example:

```
console.log("Hello Developers");
console.log('Hello Developers');
console.log('Hello \n Developers');
```

All the above statements will log valid strings in the console.

String

More Ways to Create a String Literal

```
var str1 = "Hello";
console.log(str1);
// Output:  Hello

var str2 = String("Hello");
console.log(str2);
// Output: Hello
```

String

One More Interesting Thing to Know

```
var str1 = "Hi";
console.log(str1); // logs "Hi"
console.log(str1.valueOf()); // logs "Hi"
```

Did you notice, we called valueOf() on the str1 and we assigned it a primitive value as i mentioned above primitive .
Can you identify from where we get the valueOf() function, despite it being a primitive value assigned to str1?

String

Reason

For statements of assigning primitive values to a variable like

```
var str1 = "Hi";
```

JavaScript will internally create the variable using:

```
String("Hi")
```

As a result, we have all the available functions of the `String` wrapper object.

String

Problems with string?

Problem appears when you try to create a multi line template and trying to concatenate variables and expressions , and then add a quote inside string so you need to alternate between single and double quotes that all problems solved, **ES6** came with a great Solution to save you, Welcome **Template Literals.**

Template Literals

Template Literals

A new and fast way to deal with strings is **Template Literals or Template String.**

How we were dealing with strings before ?

```
var myName = "daniyal" ;
var hello = "Hello "+ myName ;
console.log(hello); //Hello daniyal
```

Template Literals

What is Template literals ?

As we mentioned before , it’s a way to deal with strings and specially dynamic strings ; so you don’t need to think more about what’s the next quote to use single or double.

How to use Template literals

It uses a `backticks` to write string within it.

Template Literals

EXAMPLE:

```
var template = `Hello from "template literals" article , check previous one 'arrow functions'.` ;
console.log(template) // Hello from "template literals" article , check previous one 'arrow functions'.
```

Template Literals

Multiline Strings

Another great feature of template strings is that we can have strings with multiple lines for better code readability.

Template Literals

Example:

```
 const longString = `Lorem ipsum, dolor sit amet consectetur adipisicing elit.
        Excepturi laboriosam minus laudantium. Illum rem cupiditate quisquam,
        modi optio dolorem veritatis aliquam eius nam soluta.
        Dolorem adipisci natus voluptas laborum dolorum?`

// Output  "Lorem ipsum, dolor sit amet consectetur adipisicing elit.
        Excepturi laboriosam minus laudantium. Illum rem cupiditate quisquam,
        modi optio dolorem veritatis aliquam eius nam soluta.
        Dolorem adipisci natus voluptas laborum dolorum?"
```

Comments

Comments

- We normally use **Comments** to describe how and why the code works.
- Comments are used to add annotation, hints, suggestions, or warnings to JavaScript code.
- This can make it easier to read and understand.
- They can also be used to disable code to prevent it from being executed.
- This can be a valuable debugging tool.

JavaScript has two long-standing ways to add comments to code.

Comments

Comments can be single-line: starting with //

```
// generate user ticket
function triggerEvent(param){
    return ...
```

Comments

and multiline: /* ... */

```
// generate user ticket
/* function triggerEvent(param){
   return ...
} */
```

SCOPE

SCOPE

- **Scope** is a feature of javascript.
- **Scope in JavaScript** refers to the current context of code.
- Scope determines the visibility of variables, functions and other resources areas in your code stack.
- In the JavaScript language there are two types of scopes:
  - Global Scope
  - Local Scope

Global Scope

Global Scope

- When you start writing JavaScript in a document you are already in the Global scope.
- There is only one Global scope throughout a JavaScript document.
- A variable is in the Global scope if it's defined outside of a function.

```
// the scope is by default global
var userName = 'Daniyal Nagori';
```

Global Scope

```
var userName = 'John';
console.log(userName); // output 'Ameen Alam'

function func1() {
    console.log(userName); // 'userName' is accessible here and everywhere else
}

func1(); // output 'Ameen Alam'
```

Local Scope

Local Scope

- Local variables are those declared inside of a block.
- If variables defined inside a function are in the local scope.
- JavaScript has function scope: Each function creates a new scope.

```
// Global Scope
function func1() {
    // Local Scope
}
// Global Scope
```

Local Scope

Variables defined inside a function are not accessible (visible) from outside the function.

```
// Global Scope
function func1() {
    // Local Scope #1
    function func2() {
        // Local Scope #2
    }
}
// Global Scope
function func3() {
    // Local Scope #3
}
// Global Scope
```

SCOPE (Local vs Global)

There are two differences between global and local variables—where they're declared, and where they're known and can be used.

SCOPE (Local vs Global)

- A global variable is one that's declared in the main body of your code, not inside a function.
- Declared in the main code
- Known everywhere, useable everywhere

- A local variable is one that's declared inside a function.
- Declared in a function
- Known only inside the function, usable only inside the function

SCOPE (Local vs Global)

- Global variable is meaningful in every section of your code, whether that code is in the main body or in any of the functions.

- Local variable is one that's meaningful only within the function that declares it.

SCOPE – Examples

```
function myName(){
    var age = 12;
}
    myName()
    console.log(age);

// OUTPUT  Uncaught ReferenceError: age is not define
```

REASON: Variable age is defined in the function it can't be access from outside the function.

SCOPE – Examples

```
var age = 100;
function go(){
 var age = 200;
 var hair =  'black';
 console.log(age);
 console.log(hair);
}
go();
console.log(age);


// OUTPUT  200
// OUTPUT  black
// OUTPUT  100
```

SCOPE – Block Statements

- Block statements like `if` and `switch` conditions or `for` and `while` loops,
- Don't create a new scope.
- Variables defined inside of a block statement will remain in the scope they were already in.

```
if (true) {
   // this 'if' conditional block doesn't create a new scope
   var userName = 'John'; // userName is still in the global scope
}

console.log(userName); // output 'Ameen Alam'
```

SCOPE – Block Statements

ECMAScript 6 introduced the let and const keywords. These keywords can be used in place of the var keyword.

```
var userName = 'Ameen Alam';
let age = 20;
const skills = 'JavaScript';
console.log(userName); // output 'Ameen Alam'
console.log(age); // output  20
console.log(skills); // output 'JavaScript'
```

let and const keywords support the declaration of local scope inside block statements.

SCOPE – Block Statements

```
if (true) {
    // userName is in the global scope because of the 'var' keyword
    var userName = 'John Martin';
    console.log(userName); // output 'Ameen Alam'
    // age is in the local scope because of the 'let' keyword
    let age = 20;
    console.log(age); // output 20
    // skills is in the local scope because of the 'const' keyword
    const skills = 'JavaScript';
    console.log(skills); // output 'JavaScript'
}
console.log(userName); // output 'Ameen Alam'
console.log(age); // Uncaught ReferenceError: age is not defined
console.log(skills); // Uncaught ReferenceError: skills is not defined
```

Lexical Scope

SCOPE – Lexical Scope

- Lexical Scope means that in a nested group of functions
- The inner functions have access to the variables and other resources of their parent scope.
- This means that the child functions are lexically bound to the execution context of their parents.
- Lexical scope is sometimes also referred to as Static Scope.

Source: https://scotch.io/tutorials/understanding-scope-in-javascript#toc-lexical-scope

SCOPE – Lexical Scope

```
function func1() {
    var userName = 'Ameen Alam';
    // likes is not accessible here
    function parent() {
        // userName is accessible here
        // likes is not accessible here
        function child() {
            // Innermost level of the scope chain
            // userName is also accessible here
            var likes = 'Coding';
        }
    }
}
```

LET

- The let statement declares a block scope local variable, optionally initializing it to a value.
- Variables declared by let have their scope in the block for which they are defined, as well as in any contained sub-blocks.
- In this way, let works very much like var.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

CONST

// ADDING ONE MORE KEYWORD CONST
// Scopes let var const

- **CONST**
  - ES6 provides a new way of declaring a constant by using the **const** keyword.
  - The **const** keyword creates a read-only reference to a value.
  - The **const** keyword works like the let keyword.
  - The **const** keyword creates block-scoped variables whose values can't be reassigned.

SCOPE

Example:

```
if (true){
 const age = 1;
 let cool = true;
 var myAge = 23;
}
console.log(myAge);
console.log(cool);
console.log(age);
//Output 23 // var is accessible because var is a function scope.
//Output Uncaught ReferenceError: cool is not defined
//Output Uncaught ReferenceError: age is not defined
```

SCOPE

// Function Scope

```
function isCool(name){
 if (true){
     const cool = true;
 }
 console.log(cool);
}
isCool('daniyal');
//Output Uncaught ReferenceError: cool is not defined
```

REASON: Because const is block scoped.

SCOPE

```
const dog = 'snickers';

function logDog(){
  console.log(dog);
}

function go(){
  const dog = 'sunny';
  logDog();
}
go();
```

SCOPE

Another Example:

```
const dog = 'snickers';
function logDog(dog){
 console.log(dog);
}
function go(){
 const dog = 'sunny';
 logDog(dog);
}
go();
```

SCOPE

```
// Function Scope nested

function sayHi(){
 function yell(){
     console.log('hello');
 }
 yell()
}
sayHi()
```

Hoisting

What is hoisting in JS?

- Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope.
- When Javascript compiles your code, all variable declarations using **var** are hoisted/lifted to the top of their functional/local scope (if declared inside a function) or to the top of their global scope (if declared outside of a function) regardless of where the actual declaration has been made. This is what we mean by “***hoisting***”.
- Hoisting applies to variable declarations and to function declarations. Because of this, JavaScript functions can be called before they are declared.

Hoisting

EXAMPLE:

```
sayHi();


function sayHi(){
  console.log('hey');
}
```

Hoisting

```
sayHi();

function sayHi(){
  console.log('hey');
  console.log(add(1,2));
}

function add (a, b){
  return a + b;
}
```

Hoisting

// Hoisting works with regular function not other than that: Example Below

```
sayHi();
var add2 = function(a, b){
  return a + b;
}
function sayHi(){
  console.log('hey');
  console.log(add2(1,2));
}
// OUTPUT add2 is not a function
```

REASON: I will ask in class :)

LET VS VAR

The main difference is the scope difference,They both declare a variable that will hold some value.

```
var x;

let y;

console.log(x); // undefined

console.log(y); // undefined
```

DOM

- The Document Object Model, or the “DOM”, defines the logical structure of HTML documents & essentially acts as an interface to web pages.

- Through the use of programming languages such as JavaScript, we can access the DOM to manipulate websites and make them interactive.

DOM

- Whenever we write a html template and view in the browser the browser turns your html into something like that dom.
- You can actually see that in your elements tab in the browser.
- We access dom to create element, delete, update etc.

DOM - Selecting Element

```
Select any single element using querySelector
var ele = document.querySelector( 'p');
console.log(ele);

Select all element using querySelectorAll
var allEle =  document.querySelectorAll( 'p');
console.log(allEle);
```

DOM - Selecting Element

```
More specific using class
var allEle =  document.querySelectorAll('.item');
console.log(allEle);

var inner = document.querySelector('p').innerHTML;
//console.log(inner)

document.getElementById('pp').innerHTML = inner;
// console.log(document.body.innerHTML)

var b = document.querySelector('.item');
//console.log(b.querySelector('p'))
```

DOM - Selecting Element

```
function init() {

 var ele = document.querySelector('p');
 console.log(ele);

}
document.addEventListener('DOMContentLoaded', init)
```

The DOM Elements Properties

```
Check all the properties and methods on any element
var b = document.querySelector( '.item');
console.dir(b)

// Set the text content property on that element
const heading = document.querySelector( 'h2');
heading.textContent =  'hello john';
```

DOM Elements - Properties

```
  <h2>Hi
       <style>
          p{
             color: blue
           }
        </style>
   </h2>
const heading = document.querySelector( 'h2');
// Difference between innerText AND textContent
console.log(heading.innerText); // Just render the text not style elements
console.log(heading.textContent);// Render all the text within the element
```

DOM CLASSES

```
var b = document.querySelector('.item');
b.classList.add('open')
b.classList.remove('open')
//console.log(b.classList)


//Add class on runtime\

b.classList.add('round')
```

DOM CLASSES

```
// HTML
<p>hello</p>
// JS
const classEle = document.querySelector( 'p');
classEle.classList.add( 'nice');
console.log(classEle.classList);
// Style
   <style>
           .nice{ color: red; }
   </style>
```

Events

- HTML events are "things" that happen to HTML elements.
- When JavaScript is used in HTML pages, JavaScript can "react" on these events.
- Rule:
  - Get Something
  - Listen Something
  - Do Something

```
var hit = document.querySelector('.hit');

document.addEventListener('click', function(){
 console.log('hello');
})
```

Events

```
var hit = document.querySelector('.hit');
var hit1 = document.querySelector('.hit1');

hit.addEventListener('click', function(){
 console.log('hello123');
})
hit1.addEventListener('click',hey)

function hey(){
 console.log('runninggg');
}
```

Events

```
hit1.removeEventListener('click',hey)
var hitme = document.querySelectorAll('.hitMe');
hitme.addEventListener('click',function(){
 console.log('000000');
})
var hitme = document.querySelectorAll('.hitMe');
hitme.forEach(function(but){
 but.addEventListener('click',function(){
   console.dir(but);
 })
})
```

Events

```
var hitme = document.querySelectorAll( '.hitMe');
function callFunc(but) {
 but.addEventListener( 'click', function () {
   console.dir(but);
 })
}
hitme.forEach(callFunc);
```

Destructuring

Destructuring is a powerful way to create variables from values in **arrays** and **objects**, It will make your code simpler. Keep in mind that destructuring is part of the ES2015 (aka **ES6**)

Array destructuring

If you want to assign the values of an array to separate variables, you can take advantage of destructuring to achieve your goal in a simple, clean way. No need to use indexes or loops!

Destructuring

You can assign these two values to two variables:

```
var fruits  = ['Banana', 'Apple', 'Mango', 'Orange’ ];
var [firstFruit, secondFruit, thirdFruit, fourthFruit] = fruits;

console.log(firstFruit); // Outputs 'Banana'
console.log(secondFruit); // Outputs 'Apple'
console.log(thirdFruit); // Outputs 'Mango'
console.log(fourthFruit); // Outputs 'Orange'
```

Destructuring

Note that the position of the variables vs values is important here. The first new variable will always inherit the first value of the array. But what if I only need **the first 2 values as variables**, and **the rest as an array**? Yes it’s pretty simple.

```
var fruits  = ['Banana', 'Apple', 'Mango', 'Orange’ ];
var [firstFruit, secondFruit, ...rest] = fruits;

console.log(firstFruit); // Outputs 'Banana'
console.log(secondFruit); // Outputs 'Apple'
console.log(rest); // Outputs ['Mango', 'Orange’ ]
```

Destructuring

You can also set default values, if the passed array doesn’t have enough values.

```
var fruits  = [ 'Banana', 'Apple' ];
var [ firstFruit = 'Grape', secondFruit = 'PineApple', thirdFruit = 'Cherry' ] = fruits;

console.log(firstFruit); // Outputs 'Banana'
console.log(secondFruit); // Outputs 'Apple'
console.log(thirdFruit); // Outputs 'Cherry'
```

Or say you just need the first one fruit of the array, you can just declare one variable and the rest will be ignored.

```
var fruits  = [ 'Banana', 'Apple' ];
var [ firstFruit ] = fruits;
```

Destructuring

Object Destructuring

We can also use the destructuring assignment syntax to assign **object** values to variables. It’s pretty similar to what we did with arrays.

```
var contact = { name: 'daniyal', age: 23, email: 'daniyalnagori@yahoo.com' }
var {name, age, email} = contact;

console.log(name); // Outputs 'daniyal'
```

Destructuring

As you can see with the example above, we used the same names for the variables as we did for the keys of the object. However we can define variables with names that differ from the keys:

```
var contact = {name: 'daniyal', age: 23, email: 'daniyalnagori@yahoo.com'}
var {name: system, age: birth, email: address} = contact;
```

Destructuring

And just like with arrays, we can also set **default values**.

```
var contact = { name: 'daniyal', age: 23, email: 'daniyalnagori@yahoo.com' }
var { name = 'zeeshan', age, email, phone = 03170113001 } = contact;

console.log(name); // Outputs 'daniyal'
console.log(phone); // Outputs '03170113001l'
```

Spread Operator

ES6 has some great features that make working with function parameters and arrays extremely easy. Let’s take a look at two of these features: the **spread operator** and **rest parameters**

Syntax:

```
var variablename1 = [...value];
```

Spread Operator

Spread operator: allows iterables(arrays / objects / strings) to be expanded into single arguments/elements.

Rest Parameter

- ES6 has some great features that make working with function parameters and arrays extremely easy. Let’s take a look at two of these features: the **spread operator** and **rest parameters**

```
function xyz(x, y, ...z) {
  console.log(x, ' ', y); // hey hello
  console.log(z);
}
xyz("hey", "hello", "wassup", "goodmorning", "hi", "howdy")
```

Objects

Objects are in javascript is the biggest building block.

var comapny = { name: ‘panacloud’, employee: 100 }

Undefined vs null

```
const person = {
  age: 1,
  name: 'daniyal',
  sports: 'cricket'
}
person.age = {}
console.log(person);
```

Objects

```
const person = {
  age: 1,
  name: 'daniyal',
  sports: 'cricket'
}
person = {}
console.log(person);
```

```
Uncaught TypeError: Assignment to constant variable.
```

Objects

```
var person = {
  age: 1,
  name: 'daniyal',
  sports: 'cricket'
}
person = {}
console.log(person);

//If you want to freeze the object or immutable the object use:
Object.freeze()
```

Objects

```
var age = prompt('How old are you?');

var person = {
 name: 'daniyal',
 sports: 'cricket',
 age: 12
}

console.log(person['age']);
```

Objects

```
var age = prompt('How old are you?');

var person = {
 name: 'daniyal',
 sports: 'cricket',
 age: 12
}

console.log(person[age]);
```

Objects (Reference)

```
var name1 = "daniyal";
var name2 = "tariq";

if(name1 === name2){
 console.log(name1);
}
var name1 = "daniyal";
var name2 = "daniyal";

if(name1 === name2){
 console.log(name1);
}
var obj = {name: 'dani',age : 21}
var obj1 = {name: 'dani',age : 21}
if(obj1 === obj){
  console.log('12221');
}
```

Objects (Reference)

```
var obj = {name: 'dani',age : 21}
var obj1 = obj
if(obj1 === obj){
  console.log('12221');
}

var obj = {name: 'dani',age : 21}
var obj1 = {name: 'dani',age : 21}
obj = obj1;
obj.age = 31
obj1
```

Objects (Reference)

```
// To create a copy of the object or break the reference using object.assign

var obj = {name: 'dani',age : 21}
var obj1 = {name: 'dani',age : 21}

var obj = Object.assign( {}, obj1 );
obj.age = 31
obj1
```

Objects (Reference)

```
// To create a copy of the object or break the reference using spread operator

var obj = {name: 'dani',age : 21}
var obj1 = {name: 'dani',age : 21}

var obj = { ...obj1 }
obj.age = 31
obj1
```

Objects (Nested Reference)

```
var obj = {name: 'dani',age : 21}
var obj1 = {name: 'dani',age : 21, clothes: {shirt: 'white'}};

var obj = Object.assign({}, obj1);
obj.age = 31
obj.clothes.shirt = 'red'
obj1
```

OBJECTS (MERGE TWO OBJECTS)

```
var stuff = {name: 'dani',age : 21,shoes: 'nike'}
var stuff1 = {name: 'dani',age : 21, clothes: {shirt: 'white'}};

var mergeObj = {...stuff, ...stuff1 };
```

OBJECTS

```
var name1 = 'daniyal'
function doStuff(name){
 name1 = 'tariq';
 console.log(name);
}
doStuff(name1);
```

OBJECTS

```
var person = {name: 'daniyal'};

function doStuff(obj1){
 obj1.name = 'tariq';
 console.log(person);
}

doStuff(person);
```

Object Literal Shorthand

```
let cat = 'Miaow';
let dog = 'Woof';
let bird = 'Peet peet';
let someObject = {
  Cat,
  Dog,
  Bird
}
console.log(someObject);
//{  cat: "Miaow",
//  dog: "Woof",
//  bird: "Peet peet"
//}
```

Constructor

The constructor property returns a reference to the Object constructor function that created the instance object. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

All objects (with the exception of objects created with Object.create(null)) will have a constructor property. Objects created without the explicit use of a constructor function (such as object- and array-literals) will have a constructor property that points to the Fundamental Object constructor type for that object.

Constructor

```
let o = {}
o.constructor === Object // true

let o = new Object
o.constructor === Object // true

let a = []
a.constructor === Array // true

let a = new Array
a.constructor === Array // true

let n = new Number(3)
n.constructor === Number // true
```

Constructor

```
function Students(n, ag, add){
   this.name = n
   this.age = ag
   this.address = add
   this.func = function(){
       console.log("runnin function", this.name)
   }
}
var rizwan = new Students("rizwan", 20, "address.....")
console.log(rizwan.func())
var ameen = new Students("ameen", 20, "address.....")
console.log(ameen.func())
var hamza = new Students("hamza", 19, "address.....")
console.log(hamza.func())
```

Constructor

```
function Students(name, age, address){
    // let defaultVal = " default value"
    this.name = name
    this.age = age
    this.address = address
    this.func = function(){
        return `${this.name} ${this.age} dummy text`
    }
}

let hamza = new Students("Hamza", 20, 'karachi...')
console.log(hamza.func())
console.log(hamza)
```

Class

A JavaScript class is a type of function. Classes are declared with the class keyword. We will use function expression syntax to initialize a function and class expression syntax to initialize a class. We can access the [[Prototype]] of an object using the Object

Class

```
class Students {
}
let hamza = new Students()
console.log(hamza)

class Students2 {
   constructor() {
       console.log("constructor running")
   }
}
let mohsin = new Students2()
console.log(mohsin)
```

Class

```
class Students3 {
    constructor(name, age, address) {
        console.log("constructor running", name, age, address)
    }
}
// let faraz = new Students3()
let faraz = new Students3('faraz', 22, 'address...')
console.log(faraz)
```

Class

```
class Students4 {
    constructor(name, age, address) {
        this.name = name
        this.age = age
        this.address = address
        this.val = "aaa"
    }
}
let mehran = new Students4('mehran', 20, 'address...')
console.log(mehran)
```

Class

```
class Parent{
    constructor(name){
        this.name = name
    }
}

class Child extends Parent{
    constructor(name1){
        super(name1);
    }
}
let _child = new Child("ameen")
console.log(_child)
```

Class

```
class Parent {
    constructor(age){
        this.age = age
        this.name = "name"
        this.address = "Address........"
    }
}
class Child1 extends Parent{
  constructor(age){
      super(age)
  }
}
var _child1 = new Child1(50)
console.log(_child1)
```

For of loop

```
const array1 = ['a', 'b', 'c'];

for (const element of array1) {
  console.log(element);
}

// expected output: "a"
// expected output: "b"
// expected output: "c"
```

For in Loop

The JavaScript for...in statement loops through the properties of an object.

Looping through the properties of an object:

```
var company = {name: 'Panacloud', employee: 200};
var txt = '';
for (i in company){
    // console.log(company[i]);
    txt += company[i] + ' ';
}
console.log(txt);
```

Closures

```
function company(){

   var myName = "daniyal";
   console.log('company',myName);

   function employee(){
     console.log('employee',myName);
   }

   employee()

 }

 company()
```

Closures

```
function company(name){
  var companyName = name;
  return function employee(number){
    console.log(companyName + ' has ' + number + ' employees')
    return companyName + ' has ' + number + ' employees';
  }
}
 const employee = company('panacloud');
const employee1 = company('prostack');
employee(2);
employee1(4)
```

Closures

```
function sportsGame(name){
  let score = 1;
  return function abc (){
    console.log(score++);
  }
}
 var game = sportsGame('hockey');
 var game1 = sportsGame('cricket');
 game();
 game1();
```

Closures

```
function myScore (){
  var score = 1;
  function abc(){
    console.log(score++);
  }
  abc();
}
 myScore();
```

Closures

```
function myScore (){
  var score = 1;
 return function abc(){
    console.log(score++);
  }
 }

 var abc = myScore();
```

Objects.keys

```
let obj = {
    name: 'Krunal',
    education: 'IT Engineer'
} ;
console.log(Object.keys(obj)); // ["name", "education"]
```

Object.assign

```
const target = { a: 1, b: 2 };

const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(source);
// expected output: Object { b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }
```

Object.assign

```
const obj = { a: 1 };

const copy = Object.assign({}, obj);

console.log(copy); // { a: 1 }


// by spread operator
const obj = { a: 1 };

const copy = {...obj};

console.log(copy); // { a: 1 }
```

Object.value

Javascript Object values() is an inbuilt function that returns the array of the given Object’s enumerable property values. The ordering of the properties is the same as that given by the Object manually is the loop is applied to the properties.

Object.value

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false
};

console.log(Object.values(object1));
// expected output: Array ["somestring", 42, false]
```

Default Values

```
var handler = {
  get: function(target, name) {
    return target.hasOwnProperty(name) ? target[name] : 42;
  }
};
var p = new Proxy({}, handler);
p.answerToTheUltimateQuestionOfLife; //=> 42
```

Source: https://stackoverflow.com/questions/6600868/set-default-value-of-javascript-object-attributes

Set

Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set may only occur once; it is unique in the Set's collection.

Set

```
let mySet = new Set()
mySet.add(1)          // Set [ 1 ]
mySet.add(5)          // Set [ 1, 5 ]
mySet.add(5)          // Set [ 1, 5 ]
mySet.add('some text') // Set [ 1, 5, 'some text' ]
let o = {a: 1, b: 2}
mySet.add(o)
mySet.add({a: 1, b: 2})   // o is referencing a different object, so this is okay
mySet.has(1)             // true
mySet.has(3)             // false, since 3 has not been added to the set
mySet.has(5)             // true
mySet.has(Math.sqrt(25))  // true
mySet.has('Some Text'.toLowerCase()) // true
mySet.has(o)       // true
```

Local Storage

```
// Store

localStorage.setItem("name", "ameen");

// Retrieve

Let ele = document.querySelector(".result")
ele.innerText = localStorage.getItem("lastname");
```

Local Storage

```
localStorage.removeItem('myCat');


// Clear all items

localStorage.clear();
```

Session Storage

```
// Save data to sessionStorage

sessionStorage.setItem('key', 'value');


// Get saved data from sessionStorage

let data = sessionStorage.getItem('key');
```

Session Storage

```
// Remove saved data from sessionStorage

sessionStorage.removeItem('key');


// Remove all saved data from sessionStorage

sessionStorage.clear();
```

Session Storage

```
let field = document.getElementById("field");
if (sessionStorage.getItem("autosave")) {
  field.value = sessionStorage.getItem("autosave");
}
field.addEventListener("change", function() {
 sessionStorage.setItem("autosave", field.value);
})
```

TRY, CATCH, THROW

TRY, CATCH, THROW

```
// function nonExistentFunction(){
// console.log("ss")
// }
try {
  nonExistentFunction();
}
catch(error) {
  console.error(error);
  // expected output: ReferenceError: nonExistentFunction is not defined
  // Note - error messages will vary depending on browser
}
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

Conditional Operator (Ternary)

The conditional (ternary) operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and finally the expression to execute if the condition is falsy. This operator is frequently used as a shortcut for the if statement.

```
condition ? exprIfTrue : exprIfFalse
```

Conditional Operator (Ternary)

```
function getFee(isMember) {
  return (isMember ? '$2.00' : '$10.00');
}
console.log(getFee(true));
// expected output: "$2.00"

console.log(getFee(false));
// expected output: "$10.00"

console.log(getFee(1));
// expected output: "$2.00"
```

Short Circuit Evaluation (&& ||)

Logical operators are typically used with Boolean (logical) values. When they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they will return a non-Boolean value.

Short Circuit Evaluation (&& ||)

```
const a = 3;
const b = -2;

console.log(a > 0 && b > 0);
// expected output: false

console.log(a > 0 || b > 0);
// expected output: true

console.log(!(a > 0 || b > 0));
// expected output: false
```

Async JS (promises, async/await, callbacks)

ARROW FUNCTIONS

Arrow functions were introduced in ES6.
Arrow functions allow us to write shorter function syntax:

```
hello = () => "Hello World!";
```

ARROW FUNCTIONS

```
// ES5

var multiplyES5 = function(x, y) {
  return x * y;
};

// ES6

const multiplyES6 = (x, y) => { return x * y };
```

MODULES

A module is just a file. One script is one module.

Modules can load each other and use special directives **export** and **import** to interchange functionality, call functions of one module from another one:

export keyword labels variables and functions that should be accessible from outside the current module.

import allows the import of functionality from other modules.

Generator Functions

EVENTS

```
btn.addEventListener('click', function() {
  console.log(“ “)
});
```

```
const btn = document.querySelector('button');
btn.onclick = function() {
  console.log(“ “)
}
```

Functions (Built In)

Math.round()
Math.random()
Math.max
Math.floor()
Math.ceil()
ParsrFloat()
Date()
ScrollTo()

Functions (Built In)

```
element.scrollTo({
  top: 100,
  left: 100,
  behavior: 'smooth'
});
```

New