JavaScript

1. **ES6+ Features**: Start by familiarizing yourself with ES6 (ECMAScript 2015) and newer features. Learn about arrow functions, template literals, destructuring, spread/rest operators, classes, and modules.
2. **Asynchronous JavaScript**: Understand asynchronous programming deeply, especially Promises, async/await, and the Event Loop. This is crucial for handling operations like fetching data from APIs.
3. **Functional Programming**: Dive into functional programming concepts in JavaScript. Learn about higher-order functions, pure functions, and how to use array methods like **map**, **filter**, and **reduce**.
4. **Learn a Framework or Library**: Explore popular frameworks like React, Angular, or Vue.js. These frameworks leverage JavaScript in powerful ways and understanding them can significantly enhance your skill set.
5. **Modules and Bundlers**: Understand module systems like CommonJS and ES6 Modules. Familiarize yourself with bundlers like Webpack or Parcel that help manage module dependencies.
6. **Testing**: Learn testing frameworks like Jest or Mocha. Testing is crucial for maintaining code quality and catching bugs early in the development process.
7. **Build Tools**: Familiarize yourself with tools like Babel for transpiling newer JavaScript features into code that older browsers can understand, and ESLint for code linting.
8. **Web APIs and DOM Manipulation**: Understand how JavaScript interacts with the Document Object Model (DOM) and various Web APIs. This knowledge is essential for creating interactive web applications.
9. **Explore Advanced Topics**: Delve into more advanced topics like closures, prototypes, design patterns, and performance optimization. Understanding these concepts will deepen your understanding of JavaScript.
10. **Practice Regularly**: The best way to upgrade your skills is through practice. Build projects that challenge you and incorporate the concepts you've learned.
11. **Keep Learning**: JavaScript is a rapidly evolving language. Stay updated with new features, best practices, and emerging trends by following blogs, tutorials, and participating in online communities.
12. **Contribute to Open Source**: Consider contributing to open-source projects. This not only helps the community but also allows you to learn from experienced developers and improve your skills.

**Udemy**

***JavaScript Lessons***

Conceptual Aside

1. Syntax Parsers – A program that reads your code and determines what it does and if its grammar is valid
2. Execution Contexts - a wrapper to help manage the code that is running   
   {  
   i = 1;  
   execution context as the global environment  
   function a

{

J = 2;  
 another execution context

Local environment but global for inside function a.

Function b

{

K = 3;

i does have value in here since that is global variable.

}  
 K has never been initialised here;  
}

J has never been initialised here.

}

1. Lexical Environments - where something sits physically in the code you write.

Name/Value Pairs and Objects.

* A name that maps to a unique value.

An object is a collection of name/value pairs.

**Invocation –** invoke/running/calling a function

**Scope** – where the variable is available in your code

**Infix notation** (putting function like operator (+,-,\*,etc) in the middle)

+(1 , 2)

= +1 2 🡪 prefix

= 1 2+ 🡪 postfix. 🡪 just to make clear how things work but it does not work in JS likewise.

= 1 + 2 infix

Some e.g. var a = 4 > 3 returns true  
Note: Means operators are functions.

**Precedence and associativity (**which one should be taken as a priority first if have multiple operators function while associativity is like precedence to select or prioritise the operator if the operators have the same level of precedence (e.g right to the left or vice versa)

**Coercion**

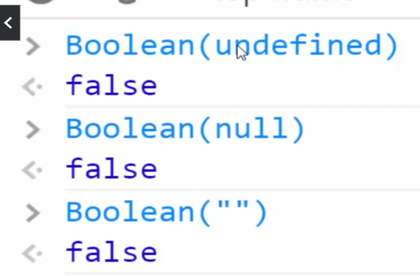
Converting a value from one type to another

**Double-equal and triple-equal**

Double equals coerce/convert the variables into the same type and perform the logic (e.g “” == 0, “” == false, both equations return true) while triple equals follow strictly to the logic and return as true only if the values are of the same type. E.g. “” === 0, “” == false, both equations return false)

Note: same goes for != and !==  
  
[**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality\_comparisons\_and\_sameness**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness) **Follow the link above to check all possible comparisons if it is true or false**

**Some of the false statements. (value 0 also represents false)**

****

Var a = 0;

If (a){

Console.log(‘it wont print this’);

}

**Cause a will be coerced to false as 0 (zero) coerced as false.**

**I**f statements check such statement be true or false.

**Use of or in default values**

****

Result: Hello Tony

Hello <Your name here>

**Reason:**

“” || “any value” returns any value. Similarly,

Null || “any value”

Undefined || “any value” returns the same “any value”.

But, “some value” || “any value” returns some value.   
e.g. true || false equals true and false || true equals true.

**Note: 0 (zero) is always false even if it is a value.**

*//new technique for the default value*

*function call(name = 'Sribika') {*

*console.log("Hello " + name);*

*}*

*call();*

*call('Biplav');*

**Window.name = window.name || “any name”;   
🡪 does not replace the value of the name variable if there is any else put “any name” as the value of the name.  
  
Note: We use this technique to check/debug whether the variable name has been used or not.**

**Objects and functions**

**Functions connected to an object are called a method.**

**Var person = new Object(); 🡪constructors notation and var person = {} 🡪 object literals; same thing where both are being used to create an object but preferred is object literals.**

**Object Literals and JSON**

var objectLiteral = {

firstName : 'Biplav',

lastName : 'Karki',

isProgrammer: true

};

console.log(JSON.stringify(objectLiteral)); // converting json into string

var jsonValue = JSON.parse('{"firstName": "Biplav", "isProgrammer": true }') // converting string json to object

console.log(jsonValue);

**Function expression and statements**

//function expressions - provide a value or results in a value

//anonymous functions that do not require name

//need to be initialised the value first to invoke

var anonymousMath = function() {

console.log(2+3);

}

console.log(anonymousMath); // prints the whole function as a string

anonymousMath(); //invokes as a function which does have value

//functional programming||first class functions||

function abc(param) {

console.log(param);

param(); // param acts as a variable and stores the passed function/object in it.

// using such variable using parenthesis can invoke the function or provide the result whatever is in the anynomous function

}

// passing function into function or passing objects into function as function is object in js.

abc( function() {

console.log("hello abc");

})

**By value and by reference**

// by value - all primitive types are by value

var a = 3;

var b;

b = a; // b is copying the value of a and storing it into a different location

a = 2; // mutate

console.log(a); // prints 2 as a has been mutated

console.log(b); // prints 3 as b copied the inital value of a which was 3

// by reference - all objects (including functions) are by references

var c = { greetings: 'Namaste'};

var d;

d = c; // d is referencing the value of a.

c.greetings = 'Hola'; //mutating

console.log(c); //prints hola as it is changed

console.log(d); // prints hola which is same to c because both variable is pointing the same location

// by reference (even as parameters)

//anonymous function

var greetobj = function(c) {

c.greetings = 'Namaskar';

}

greetobj(c);

console.log(c);

console.log(d);

//stated functions

changeGreeting(c);

function changeGreeting(obj){

obj.greetings = "K chha";

}

console.log(c);

console.log(d);

// equals operator sets up new memory space (new address)

c = {greeting:"How you doin"};

console.log(c); //prints How you doing as C sets up new value on new location

console.log(d); // prints K chha

**Note: mutate 🡪 to make change**

**Note: Inside the object: If the value is primitive, it’s called the property and if the value is a function, it’s called the method.**

Var a = {

this.case = ‘ case’; // cannot use such syntax in object literals

}

*Note: functions can be invoked, whereas regular objects can't.*

*Note: ‘this’ object inside the inner function ( function inside the function) will point to the Window or global object.*

*Note: if we create a method/function(1) inside the object, we need to create another object (that, self, etc) that will point to the same location as the ‘this’ object but the ‘self’ object inside the new inner function(2) which is inside the function(1) will not point to the global object.*

A screen shot of a computer program

Description automatically generated

*Note: JavaScript array can be a collection of any data types like int, Boolean, objects, functions, and strings.*

*A computer code with colorful text

Description automatically generated*

*Output: Hello Biplav*

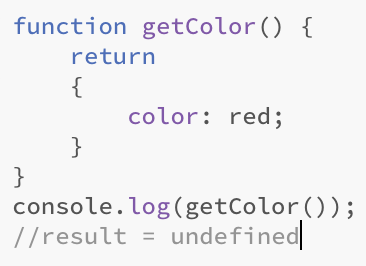
**Arguments**

The parameters you passed to the functions called arguments.

****

*Note: there is no function overloading in js since the functions are objects.*

***Note: In js, the semicolon is not optional but the js engine does it for you. Whenever you put enter (carriage return) on the return statement, the js engine (syntax parser) automatically inserts a semicolon for you. This could cause problems. Example:***

****

Because the engine inserts a semicolon after the return automatically which terminates the function and returns undefined. So, it is always better to put curly braces {} after a space instead of using enter (carriage return) to write the curly braces on the next line.

**IIFE (**Immediately invoke function expressions**)**

**A computer screen shot of a computer code

Description automatically generated**

A blue screen with black text

Description automatically generated

*NOTE: if we don’t use parenthesis () then the parsers will throw an error as it expects some name if the statement starts with function. If we write anything inside brackets, the parsers take them as expressions, not statements.*



*NOTE: those three will be taken as one line of code by parsers but do not throw errors even if they are not doing anything because the parsers think those are just expressions. The upper classic IIFE is a similar technique to shorten code.*



***When the nameless function creates an execution context then the execution context created by the greet function will be removed but the value remains somewhere in the memory. Such value can be utilised in the execution context of a nameless function using the scope technique. Now, the execution context of both functions results in a single i.e. closure.***

******

*function buildFunction(){*

*var arr = [];*

*for(i = 0; i < 3; i++){*

*arr.push(*

*(function(j){*

*return function(){*

*console.log('Hello ' + j); // when the system tries to access the value of j, it does not need to go out up until the for loop. It will just go out the execution context created by this function 🡪 (function(j){})*

*}*

*}(i))*

**Function Factory**

******

A function factory is a higher-order function that produces or returns another function based on certain conditions or parameters provided to it. It's a concept in functional programming where functions are used to create other functions dynamically.They're commonly used in scenarios where you need to generate multiple functions with similar behaviour but different configurations or data.

**Closers, function expressions and call-backs**

**Call(), apply() and bind()**

****

*To get access to data, methods, variables, etc, of another execution context from the global execution context. ()*