JavaScript

JS Functions: -

JavaScript functions are defined with the function keyword.

```
function functionName(parameters) {
  // code to be executed
}
```

It is a declared function and will get executed when invoked(called).

Function Expressions: -

A function expression can be stored in a variable:

```
const x = function (a, b) {return a * b};
let z = x(4, 3); // 12
```

The function above is actually an anonymous function (a function without a name). They are always invoked (called) using the variable name.

Function() Constructor: -

```
const myFunction = new Function("a", "b", "return a * b");
let x = myFunction(4, 3); // 12
```

Function Hoisting: -

functions can be called before they are declared.

```
myFunction(5);
function myFunction(y) {
  return y * y;
}
```

Self-Invoking Functions: -

It is invoked (started) automatically, without being called. Function expressions will execute automatically if the expression is followed by ().

```
(function () { // you cannot call it
  let x = "Hello!!";
  console.log(x);
})(); // anonymous self-invoking function
```

Functions are Objects: -

JavaScript functions have both properties and methods.

arguments.length: -

```
function myFunction(a, b) {
  return arguments.length; // 2
}
```

The **toString()** method returns the function as a string.

```
function myFunction(a, b) {
  return a * b;
}
let text = myFunction.toString(); // function myFunction(a, b) { return a * b; }
```

Note: -

A function defined as the property of an object, is called a method to the object.

A function designed to create new objects, is called an object constructor.

Arrow Functions: -

Arrow functions allow a short syntax for writing function expressions.

```
const x = (x, y) => x * y;
x(5,6); // 30
```

Arrow functions do not have their own this. They are not well suited for defining object methods. It are not hoisted. They must be defined before they are used. Using const is safer than using var, because a function expression is always constant value.

You can only omit the return keyword and the curly brackets if the function is a single statement. It might be a good habit to always keep them.

Function Parameters and Arguments: -

Function parameters are the names listed in the function definition.

Function arguments are the real values passed to (and received by) the function

Parameter Rules

JavaScript function definitions do not specify data types for parameters. It do not perform type checking on the passed arguments and also do not check the number of arguments received.

If a function is called with missing arguments (less than declared), the missing values are set to undefined.

Default Parameter Values: -

If y is not passed or undefined, then y = 10

```
function myFunction(x, y = 10) {
  return x + y;
}
myFunction(5); // 15
```

Function Rest Parameter: -

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}
let x = sum(4, 9, 16, 25, 29, 100, 66, 77); // 326
```

The Arguments Object: -

The argument object contains an array of the arguments used when the function was called (invoked).

```
x = findMax(1, 123, 500, 115, 44, 88); // 500
function findMax() {
  let max = -Infinity;
  for (let i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}
```

If a function is called with too many arguments (more than declared), these arguments can be reached using the arguments object.

JavaScript arguments are passed by value. The function only gets to know the values, not the argument's locations. If a function changes an argument's value, it does not change the parameter's original value. Changes to arguments are not visible (reflected) outside the function.

Objects are Passed by Reference: -

If a function changes an object property, it changes the original value.

Changes to object properties are visible (reflected) outside the function.

Invoking a JavaScript Function: -

```
function myFunction(a, b) {
  return a * b;
}
myFunction(10, 2);      // call or invoke
```

The function above does not belong to any object. But in JavaScript there is always a default global object.

In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.

myFunction() and window.myFunction() is the same function.

What is this?

The this keyword refers to an object. this is not a variable. It is a keyword.

```
let x = myFunction();             // x will be the window object
function myFunction() {
  return this;
}
```

```
const myObject = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
myObject.fullName();         // Will return "John Doe"
```

```
const myObject = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this;
  }
```

```
}
// This will return [object Object] (the owner object)
myObject.fullName();
```

Invoking a Function with the Constructor: -

```
// This is a function constructor:
function myFunction(arg1, arg2) {
  this.firstName = arg1;
  this.lastName  = arg2;
}
// This creates a new object
const myObj = new myFunction("John", "Doe");
// This will return "John"
myObj.firstName;
```

The JavaScript call() Method: -

It can be used to invoke (call) a method with an object as an argument (parameter).

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}
const person2 = {
  firstName:"Mary",
  lastName: "Doe"
}
// This will return "John Doe":
person.fullName.call(person1);
```

The call() method can accept arguments

The JavaScript apply() Method: -

The apply() method is similar to the call() method

The difference is:

The call() method takes arguments separately.

The apply() method takes arguments as an array.

The apply() method accepts arguments in an array.

```
person.fullName.apply(person1, ["Oslo", "Norway"]);
person.fullName.call(person1, "Oslo", "Norway");
```

Function Borrowing: -

With the bind() method, an object can borrow a method from another object.

```
const person = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
const member = {
  firstName:"Hege",
  lastName: "Nilsen",
}
let fullName = person.fullName.bind(member); // Hege Nilsen
```

JavaScript Closures: -

JavaScript variables can belong to the local scope or the global scope

Local Variables: -

A local variable is a "private" variable defined inside a function.

```
function myFunction() {
  let a = 4;
```

```
  return a * a;
}
```

A local variable can only be used inside the function where it is defined. It is private and hidden from other functions and other scripting code.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

Global Variables: -

A global variable is a "public" variable defined outside a function.

```
let a = 4;
function myFunction() {
  return a * a;
}
```

Global variables can be used (or changed) by all scripts in the page.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Global variables live until the page is discarded, like when you navigate to another page or close the window.

A Counter Dilemma: -

you want to use a variable for counting something, and you want this counter to be available to everyone (all functions). You could use a global variable, and a function to increase the counter.

```
// Initiate counter

let counter = 0;

// Function to increment counter

function add() {

  counter += 1;

}

// Call add() 3 times

add();

add();

add();

// The counter should now be 3
```

Now, Any code on the page can change the counter, without calling add(). The counter should be local to the add() function, to prevent other code from changing it

```
// Initiate counter

let counter = 0;

// Function to increment counter

function add() {

  let counter = 0;

  counter += 1;

}

// Call add() 3 times

add();

add();

add();

// The counter should now be 3. But it is 0
```

It did not work because we display the global counter instead of the local counter. We can remove the global counter and access the local counter by letting the function return it.

```
// Function to increment counter

function add() {

  let counter = 0;

  counter += 1;

  return counter;

}

let x= 0;

// Call add() 3 times

x = add();

x = add();

x = add();

// The counter should now be 3. But it is 1.
```

It did not work because we reset the local counter every time we call the function

JavaScript Nested Functions: -

A JavaScript inner function can solve this.

```
function add() {
  let counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;
}

// it will give 1.
```

It could have solved the counter dilemma, if we could reach the plus() function from the outside. Also need to find a way to execute counter = 0 only once.

```
function myCounter() {

  let counter = 0;

  return function() {

    counter++;

    return counter;

  };

}

const add = myCounter();

add();

add();

add();

// the counter is now 3
```

The variable add is assigned to the return value of a function. The function only runs once. It sets the counter to zero (0), and returns a function expression.

This is called a closure.

A closure is a function that has access to the parent scope, after the parent function has closed.

It has been used to Create private variables, Preserve state between function calls, Simulate block-scoping before let and const existed, Implement certain design patterns.

JavaScript Maps: -

A Map holds key-value pairs where the keys can be any datatype. A Map remembers the original insertion order of the keys.

The new Map() Method: -

```
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

The set() Method: -

```
// Create a Map

const fruits = new Map();

// Set Map Values

fruits.set("apples", 500);

fruits.set("bananas", 300);

fruits.set("oranges", 200);
```

The set() method can also be used to change existing Map values.

The get() Method: -

```
fruits.get("apples");    // Returns 500
```

Maps are also Objects.

Differences between JavaScript Objects and Maps: -

Object	Map
Not directly iterable	Directly iterable
Do not have a size property	Have a size property
Keys must be Strings (or Symbols)	Keys can be any datatype
Keys are not well ordered	Keys are ordered by insertion
Have default keys	Do not have default keys

Map.size: -

The size property returns the number of elements in a map.

```
fruits.size; // 3
```

Map.delete(): -

The delete() method removes a map element.

```
fruits.delete("apples");
```

Map.clear(): -

The clear() method removes all the elements from a map.

```
fruits.clear();
```

Map.has(): -

The has() method returns true if a key exists in a map.

```
fruits.has("apples");
```

Map.forEach(): -

The forEach() method invokes a callback for each key/value pair in a map.

```
// List all entries
let text = "";
fruits.forEach (function(value, key) {
  text += key + ' = ' + value;
})
```

Map.entries()

The entries() method returns an iterator object with the [key,values] in a map.

```
// List all entries
let text = "";
for (const x of fruits.entries()) {
  text += x;
}
```

Map.keys(): -

The keys() method returns an iterator object with the keys in a map.

```
// List all keys
let text = "";
for (const x of fruits.keys()) {
  text += x;
}
```

Map.values(): -

The values() method returns an iterator object with the values in a map.

```
// List all values
let text = "";
for (const x of fruits.values()) {
  text += x;
}
```

Being able to use objects as keys is an important Map feature.

```
// Create Objects
```

```
const apples = {name: 'Apples'};

const bananas = {name: 'Bananas'};

const oranges = {name: 'Oranges'};

// Create a Map

const fruits = new Map();

// Add new Elements to the Map

fruits.set(apples, 500);

fruits.set(bananas, 300);

fruits.set(oranges, 200);
```

The key is an object (apples), not a string ("apples")

Map.groupBy(): -

The Map.groupBy() method groups elements of an object according to string values returned from a callback function.

```
// Create an Array

const fruits = [

  {name:"apples", quantity:300},

  {name:"bananas", quantity:500},

  {name:"oranges", quantity:200},

  {name:"kiwi", quantity:150}

];

// Callback function to Group Elements

function myCallback({ quantity }) {

  return quantity > 200 ? "ok" : "low";

}

// Group by Quantity

const result = Map.groupBy(fruits, myCallback);

Map(2) {

  'ok' => [

    { name: 'apples', quantity: 300 },
```

```
    { name: 'bananas', quantity: 500 }
  ],
  'low' => [
    { name: 'oranges', quantity: 200 },
    { name: 'kiwi', quantity: 150 }
  ]
}
```

Object.groupBy() vs Map.groupBy(): -

Object.groupBy() groups elements into a JavaScript object.

Map.groupBy() groups elements into a Map object.

JavaScript Sets: -

JavaScript Set is a collection of unique values.

Each value can only occur once in a Set.

The values can be of any type, primitive values or objects.

The new Set() Method: -

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Create a Set

const letters = new Set();

// Add Values to the Set

letters.add("a");

letters.add("b");

letters.add("c");
```

If you add equal elements, only the first will be saved

Listing the Elements

```
// Create a Set

const letters = new Set(["a","b","c"]);

// List all Elements

let text = "";
```

```
for (const x of letters) {
  text += x;
}
```

Sets are also Objects.

The has() Method: -

The has() method returns true if a specified value exists in a set.

```
// Create a Set
const letters = new Set(["a","b","c"]);
// Does the Set contain "d"?
answer = letters.has("d"); // false
```

The forEach() Method: -

```
// Create a Set
const letters = new Set(["a","b","c"]);
// List all entries
let text = "";
letters.forEach (function(value) {
  text += value;
})
```

The values() Method: -

```
// Create a Set
const letters = new Set(["a","b","c"]);
// Get all Values
const myIterator = letters.values();
// List all Values
let text = "";
for (const entry of myIterator) {
  text += entry;
}
```

The keys() Method: -

A Set has no keys, so keys() returns the same as values()

The entries() Method: -

the entries() method returns [value,value]

```
// Create a Set
const letters = new Set(["a","b","c"]);
// Get all Entries
const myIterator = letters.entries();
// List all Entries
let text = "";
for (const entry of myIterator) {
  text += entry;
} // a,ab,bc,c
```

High Level Design (HLD) and Low Level Design (LLD): -

High-Level Design (HLD) and Low-Level Design (LLD) are two critical phases in the software development lifecycle. They help in translating requirements into a structured and detailed plan for implementation.

High-Level Design (HLD): -

It provides a big-picture view of the system. It focuses on the overall architecture, modules, and interactions between components without diving into implementation details.

It focuses on System Architecture, Modules/Subsystems, Data Flow, Technology Stack, Interfaces.

It provides a blueprint for the system.

It ensures all stakeholders (developers, testers, managers) have a common understanding of the system.

It helps to identify potential risks and constraints early.

Define "what" the system will do

Example: -

For an e-commerce application, the HLD might include:

Modules: User Management, Product Catalog, Shopping Cart, Payment Gateway.

Architecture: Microservices or Monolithic.

Data Flow: How user data flows from the frontend to the backend and database.

Technology Stack: React for frontend, Node.js for backend, MongoDB for database.

Low-Level Design (LLD): -

Low-Level Design provides detailed specifications for implementing the system. It focuses on the internal logic, classes, functions, and data structures of each module.

It focuses on Class Diagrams, Database Schema, Algorithm Design, Sequence Diagrams, Error Handling, State Diagrams, Code-Level Details.

It provides a detailed roadmap for developers to write code.

It ensures consistency and maintainability in the implementation.

To address edge cases and optimize performance.

Define "how" the system will do it

Example: -

For the "Shopping Cart" module in an e-commerce application, the LLD might include:

Class Diagram: Cart, Product, User classes with attributes and methods.

Database Schema: Tables like cart_items, products, and users with relationships.

Sequence Diagram: Steps for adding a product to the cart, including API calls and database updates.

Algorithm: Logic for calculating discounts or taxes.

Error Handling: What happens if a product is out of stock, or the cart is empty.