Table Of Contents

1. JS Variables……………………………………………
2. JS Naming Conventions……………………………….
3. JS Reserved Words…………………………………….
4. JS Data Types………………………………………….
5. JS Numbers…………………………………………….
6. JS Identifiers…………………………………………
7. JS String……………………………………………….
8. JS Boolean…………………………………………….
9. JS Undefined………………………………………….
10. JS Null…………………………………………………
11. JS Type Conversion……………………………………
12. JS Operator……………………………………………
13. JS Statement…………………………………………..
14. JS Math………………………………………………..
15. JS Date…………………………………………………
16. JS Comments…………………………………………..
17. JS Conditions………………………………………….
18. JS Logical Shorthands………………………………
19. JS Loops………………………………………………
20. JS Brackets……………………………………………
21. JS Array……………………………………………….
22. JS Object √……………………………………………
23. JS Global Object……………………………………….
24. JS JSON Object………………………………………..
25. JS Functions √…………………………………………..
26. JS Functional Programming……………………………
27. JS Scope……………………………………………….
28. JS Engine……………………………………………….
29. JS Interpret vs Compile……………………………….
30. JS Execution Context √…………………………………
31. JS OOP √……………………………………………
32. JS Destructuring…………………………………….

**JS Variables**

* Kinds of Variable:
  + var: The ‘var’ keyword was traditionally used to declare variable in JavaScript
  + let: The ‘let’ keyword allows you to create variable with block scope.
  + const: The ‘const’ keyword is used to create variables with block scope, just like ‘let’. However, once you assign a value to a ‘const’ variable you cannot assign to a new value.

**JS Naming Conventions**

* Kinds of Conventions:
  + camelCase
  + PascalCase
  + snake\_case
  + UPPER\_CASE
  + Prefixes and Sufixes
  + Meaningful and Descriptive Names

**JS Reserved Words**

* Version of Reserved Words:
  + Keyword (ES6 and Earlier)
  + Keyword (ES6 and Later)
  + Reserved (Future Keywords)

**JS Data Types**

* Kinds of Data Types:
  + Primitive Data Types:
    - number
    - string
    - boolean
    - null
    - undefined
    - NaN
    - symbol(ES6)
  + Complex Data Types(Reference Types):
    - object
    - array
    - date
    - regular expression

**JS Number**

* Kinds of Object:
  + Number()
* Kinds of Methods:
  + number.toFixed(digits)
  + number.toPrecision(digits)
  + number.toString()
  + parseInt(number)
  + parseFloat(number)
  + isNaN(value)
  + isFinite(number)
  + isInteger(number)
  + isSafeInteger(number)
  + Number.MAX\_VALUE
  + Number.MIN\_VALUE
  + Number.MAX\_SAFE\_INTEGER
  + Number.MIN\_SAFE\_INTEGER
* Kinds of Properties:
  + Number.prototype
    - Example:
      * Number.prototype.isOdd = function(){}

**JS Identifiers**

* Kinds of Identifiers:
  + Camel Case: Use for variable & function
    - Example:
      * var myFirstName = ‘Jagadish Chakma’
  + Pascal Case: Use for class name:
    - Example:
      * class Person{}
* All Caps: Use for constant variables:
  + Example:
    - const PI = 3.14159
    - const MAX\_ATTEMPTS = 5
    - const CUSTOM = ‘custom fixed value that will be not changed’
* Underscore Prefix: Use fo private variables:
  + Example:
* let \_userPass = ‘Password’

**JS Strings**

* Kinds of Strings:
  + single quotes (‘’)
  + double quotes (“”)
  + back ticks (``)
* Escape Characters:
  + single quote: \’
  + double quote: \”
  + backslash: \\
  + new line: \n
  + tab: \t
  + backspace: \b
* String Length Property:
  + string.length
* String Concatenation:
  + string+string+string….+N
* Template Literals:
  + `stri${n}g`
* String Literal(Object):
  + String(‘string’)
* String Methods:
  + string.charAt(index)
  + string.charCodeAt(index)
  + string.concat(string2,...stringN)
  + string.indexOf(searchValue,fromStartIndex)
  + string.lastIndexOf(searchValue,uptoindex)
  + string.startsWith(searchString,position)
  + string.endsWith(searchString)
  + string.includes(searchValue,fromIndex)
  + string.toLowerCase()
  + string.toUpperCase()
  + string.trim()
  + string.substr(startIndex, endIndex)
  + string.slice(startIndex, endIndex)
  + string.replace(searchValue, replaceValue)
  + string.split(separator, limit)
  + string.match(value)
  + string.search(regexp)

**JS Boolean**

* Kinds of Boolean:
  + true
  + false

**JS Undefined**

* Declare but not assign any value.

**JS Null**

* Declare but intentionally announcing no have value assigning null value.

**JS Type Conversion**

* Kinds of Type Conversion:
  + String to Number:
    - Number(‘100’)
* Number to String:
  + String(10)
* Any to Boolean:
  + Boolean(any)
* Falsy/Truthy Value:
  + Falsy Value:
    - “”
    - 0
    - false
    - null
    - undefined
    - NaN
  + Truthy Value:
    - 1
    - true
    - all without falsy

**JS Operator**

* Kinds of JS Operator:
  + Arithmetic Operator:
    - addition(+)
    - subtraction(-)
    - multiplication(\*)
    - division(/)
    - modulus/reminder of division(%)
    - exponentiation(\*\*)
  + Assignment Operator:
    - assignment(=)
    - addition assignment(+=)
    - subtraction assignment(-=)
    - multiplication assignment(\*=)
    - division assignment(/=)
    - modulus assignment(%=)
    - exponentiation assignment(\*\*=)
  + Comparison Operator:
    - Equality(==)
    - Inequality(!=)
    - Strict equality(===)
    - Strict inequality(!==)
    - Greater than(>)
    - Less than(<)
    - Greater than or equal to(>=)
    - Less than or equal to(<=)
* Logical Operator:
  + Logical AND(&&)
  + Logical OR(||)
  + Logical NOT(!)
* Increment/Decrement Operator:
  + Increment by 1(++)
    - post-increment
    - pre-increment
  + Decrement by 1(--)
    - post-decrement
    - pre-decrement
* Conditional(Ternary) Operator:
  + condition ? expr1 : expr2
* Type Operator:
  + typeof
  + instanceof
* Comma Operator:
  + ,
* Spread Operator:
  + …
* Delete Operator:
  + delete

**JS Statement**

* Kinds of Statement:
  + Expression Statement
  + Declaration Statement
  + Conditional Statement
  + Loop Statement
  + Control Flow Statement
  + Try-Catch Statement
  + Debugger Statement
  + Function Statement

**JS Math Function**

* Kinds of Math Function:
  + Math.abs()
  + Math.ceil()
  + Math.floor()
  + Math.round()
  + Math.max()
  + Math.min()
  + Math.pow(base,exponent)
  + Math.sqrt()

**JS Date**

* Date Components:
  + Creating a Date Object:
    - new Date();
  + Getting Date and Time Components:
    - date.getFullYear()
    - date.getMonth()
    - date.getDate()
    - date.getHours()
    - date.getMinutes()
    - date.getSeconds()
    - date.getMilliSeconds()
    - date.getDay()
  + Setting Date and Time Components:
    - date.setFullYear()
    - date.setMonth()
    - date.setDate()
    - date.setHours()
    - date.setMinutes()
    - date.setSeconds()
    - date.setMilliSeconds()
  + Formatting the Date as a String:
    - date.toDateString()
    - date.toTimeString()
    - date.toLocaleString()

**JS Comments**

* Kinds of Comment:
  + Single Line Comment: //
  + Multi Line Comment: /\* \*/

**JS Conditions**

* Kinds of Conditions:
  + ‘if’ statement
  + ‘if...else’ statement
  + ‘if...else if...else’ statement
  + switch statement
  + Ternary operator
    - condition ? expr1 : expre2

**JS Logical Shorthand**

* Kinds of Logical Shorthand:
  + Short-circuit evaluation with ‘||’ (OR):
    - value || default Value
  + Short-circuit evaluation with ‘&&’ (AND):
    - value && some function

**JS Loops**

* Kinds of Loops:
  + ‘for’ loop
  + ‘while’ loop
  + ‘do-while’ loop
  + ‘for...in’ loop
  + ‘for...of’ loop
* JS Loops Helper:
  + break
  + continue
* JS Loops Characteristics:
  + infinity
  + nested loop

**JS Brackets**

* Kinds of Brackets:
  + Parenthesis Brackets ()
  + Square Brackets []
  + Curly Brackets {}

**JS Array**

* Kinds of Array:
  + Indexed Array:[]
  + Multidimensional Array:[[],[],[]]
* Ways of Declaration of Array:
  + Literal Way: [],[[],[],[]]
  + Constructor Way: new Array(1,2,3), Array([],[],[],[])
* Array Default Loop:
  + for...of loop
* Array Data Types:
  + object: typeof(array)
* Array Spread Operator:
  + ...array
* Array Traversing & Iteration:Deep Dive:
  + **Traversing**: Array traversing involves visiting each element of an array sequentially, usually to perform some action on each element. In this approach, you explicitly control the iteration and access the elements one by one.
  + **Iteration**: Array iteration refers to the process of going through each element of an array automatically without explicit control, usually to apply a function to each element.
* Array Properties:
  + length: array.length
* Array Methods With Categorized:
  + **Adding or Removing or Replacing Elements:**
    - push(): Add elements to the end of an array.
    - pop(): Remove and return the last element of an array.
    - unshift(): Add elements to the beginning of an array.
    - shift(): Remove and return the first element of an array.
    - splice(startIndex, deleteCount, addElements): Add or remove or replace elements from any position in an array.
    - fill(fillSingleVlue, startIndex, endIndex): Fill the specific index amounts of elements of an array with a single value.
* All Array Methods:
  + - push()
    - pop()
    - shift()
    - unshift()
    - concat()
    - slice()
    - splice()
    - forEach()
    - map()
    - filter(*callBack(value, index, originalArray)*) √√√
    - reduce()
    - reduceRight()
    - find()
    - findIndex()
    - some()
    - every()
    - sort()
    - reverse()
    - includes()
    - indexOf()
    - lastIndexOf()
    - isArray()
    - flat()
    - flatMap()
    - from()
    - of()
    - join()
    - toString()
    - toLocaleString()
    - keys()
    - values()
    - entries()
    - copyWithin()
    - fill()
    - shift()
    - unshift()
    - at()
    - every()
    - filter()
    - find(*callBack(element, index, thisArg)*) √√√
    - findIndex(*callBack(element, index, thisArg)*) √√√
    - forEach(*callBack(value, index, originalArray)*) **√√√**
    - map(*callBack(value, index, originalArray)*) **√√√**
    - reduce(*callBack(prev,next,index,originalArray), initialValue*) √√√
    - reduceRight()
    - some()
    - flat()
    - flatMap()
    - flatMap()
    - concat()
    - includes()
    - indexOf()
    - lastIndexOf()
    - join()
    - keys()
    - values()
    - entries()
    - toString()
    - toLocaleString()
    - copyWithin()
    - fill()
    - filter()
    - find()
    - findIndex()
    - forEach()
    - map()
    - reduce()
    - reduceRight()
    - some()
    - flat()
    - flatMap()
    - includes()
    - indexOf()
    - lastIndexOf()
    - join()
    - keys()
    - values()
    - entries()
    - toString()
    - toLocaleString()
    - copyWithin()
    - fill()
    - filter()
    - find()
    - findIndex()
    - forEach()
    - map()
    - reduce()
    - reduceRight()
    - some()
    - flat()
    - flatMap()
    - includes()
    - indexOf()
    - lastIndexOf()
    - join()
    - keys()
    - values()
    - entries()
    - toString()
    - toLocaleString()
    - copyWithin()
    - fill()

**JS Object**

* Ways To Declare of Object: √√√
  + Literal Way: {}
  + Factory Function Way: ()=>{return{}}
  + Function Constructor Way: function Person(){}
  + ES6 Class Syntax: class Person{}
* Points of Object:
  + Access,Update,Add Properties: object.property,object[‘property’]
  + Access Methods: object.method()
  + Destructuring: {name,age} = person
  + Default Loop: for...in
  + Spread operator: {...object}
  + in operator: (property in person)
  + Nested Object: {name: ‘Jagadish, degree: {}’}
  + Object Prototypes and Inheritance: JavaScript uses prototypes for inheritance. Objects can inherit properties and methods from other objects.
  + JSON: JavaScript Object Notation (JSON) is a popular format for data exchange client to server that is based on the structure of JavaScript objects.
* ‘delete’ operator: modern js does not use this operator for downside potential.
* Properties
  + constructor: Returns a reference to the constructor function that created the instance.
  + prototype: Allows you to add properties and methods to all instances of an object.
* Methods: √√√
  + Object.keys(obj) √√√
    - *Returns an array of the object's enumerable property names.*
  + Object.values(obj) √√√
    - *Returns an array of the object's enumerable property values.*
  + Object.entries(obj) √√√
    - *Returns an array of arrays containing key-value pairs of the object's enumerable properties.*
  + Object.fromEntries(array) √√√
    - converts an array of key-value pairs into an object.
  + Object.assign(target, ...source1,source2) √√√
    - *This method is used to copy the values of all enumerable properties from one or more source objects to a target object. It returns the modified target object.*
  + Object.create(object, addProperties) √√√
    - *This method creates a new object with the specified prototype object and optionally adds additional properties to the created object.*
  + Object.defineProperty(obj, prop, descriptor)
    - *is a method in JavaScript used to define a new property or modify an existing property on an object.*
  + Object.freeze(obj) √√√
    - *is a method in JavaScript that is used to prevent any changes to an object.*
  + Object.seal(obj) √√√
    - *is a method in JavaScript that is used to prevent the addition and deletion of properties to/from an object.*
    - *However, it allows the modification of existing properties.*
  + Object.getOwnPropertyNames(obj) √√√
    - *is a method in JavaScript that returns an array of all the own (non-inherited) property names of the specified object.*
  + object.hasOwnProperty(‘propertyName’) √√√
    - *Checks if the object has a specific property (not inherited).*

**JS Global Object**

* Points:
  + Definition: In the context of programming, the term "global" refers to something that is accessible from anywhere in your code, regardless of the scope or context. When something is global, it means it's available throughout the entire program without any need for explicit importing, including, or special referencing.
  + Let, const and inside scope not a part of global properties or methods.
* Browser Environment (Web Browsers):
  + In web browsers, the global object is usually referred to as the window object. Any variable or function declared in the global scope becomes a property of the window object.
  + window.property
  + window.method()
* Node.js Environment:
  + In the Node.js environment, the global object is referred to as the global object. Similarly to the browser's window object, variables declared in the global scope become properties of the global object.
  + global.property
  + global.method()
* Other(All) Environments:
  + In other environments, there might be different names for the global object, but the concept remains the same: it provides a way to access global variables and functions from any part of your code.
  + globalThis.property
  + globalThis.method()

**JS JSON Object**

* Definition
  + JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It's often used for transmitting data between a server and a web application, and it's also used as a configuration file format.
* Methods:
  + JSON.parse(jsonString): Parses a JSON string and returns a JavaScript object.
  + JSON.stringify(jsonObject): Converts a JavaScript object into a JSON string.

**JS Functions**

* Functions √√√
  + A function is a reusable block of code that performs a specific task. It takes inputs(parameter) and produces outputs(return value).
* Function Call √√√
  + Calling a function means executing its code with specific inputs.
* Parameters and Arguments √√√
  + Parameters are placeholder in a function’s definition. Arguments are actual values passed to the function when it’s called.
* Return Statement √√√
  + Functions can produce a result using the ‘return’ statement.
* Arrow Functions(ES6) √√√
  + const add = (a,b)=>{return a+b}
  + const add = (a,b)=>a+b
* Anonymous Functions √√√
  + Functions without a name, often use as arguments in other functions.
  + Use properties method in objects.
* Higher-Order Functions √√√
  + Functions that take other functions as arguments or return them.
* Callback Functions √√√
  + Functions passed as arguments to another function, to be executed later.
* Closures √√√
  + Functions remember their outer scope after they’re executed.
* Scope √√√
  + Defines where variables are accessible.
  + Call as – local scope
* IIFE (Immediately Invoked Function Expression) √√√
  + A function that’s executed right after being defined.
* Function Context(this) √√√
  + Refers to the object the function is executing in.
* Currying Function√√√
  + Transforming a function that takes multiple arguments into a sequence of functions that each take a single argument.
* Rest Parameters √√√
  + ...args: *Rest parameters allow you to gather all remaining arguments into a single array.*
* ‘this’ Keywords √√√
  + ‘this’ keywords added a new object’s provided function.
* Recursive Function √√√
  + A recursive function is a function that calls itself as part of its execution.
* Promises(ES6)
  + Used for handling asynchronous operations in a more structured way.
* Async/Await (ES8)
  + A syntax for writing asynchronous code that looks more synchronous.
* Function Composition √√√
  + Function composition in JavaScript involves combining multiple functions to create a new function.
  + This allows you to chain functions together, where the output of one function becomes the input of the next.
  + Function composition is a fundamental concept in functional programming and can lead to more modular and reusable code.
* Special Object √√√
  + arguments: *An array-like object available inside a function that holds all the arguments passed to the function. It allows you to access arguments by index and determine the number of arguments dynamically.*
* Default Parameters √√√
  + Default parameters allow you to specify default values for function parameters.
* Function Properties √√√
  + **name:** *Returns the name of the function as a string.*
  + **length:** *Returns the number of parameters expected by the function.*
  + **arguments**: An object {}
  + **caller:**
  + **prototype:** *Allows you to add properties and methods to the prototype of the function.*
* Function Methods √√√
  + **apply(obj,[argsArray]):** *Invokes a function and sets the this value and arguments using an array or an array-like object.*
  + **call(obj, arg1,arg2...):** *Similar to apply(), but arguments are passed individually.*
  + **bind(obj, arg1, arg2...):** *Creates a new function with a specific this value and, optionally, preset arguments.*
  + **toString():** *Returns the source code of the function as a string.*

**JS Functional Programming**

* Definition

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

* Type of Functional Programming
  + Pure functions
    - A pure function is a function that, given the same input, always produces the same output and has no side effects.
    - Pure functions are easier to test, reason about, and parallelize.
  + First Class Functions
    - JavaScript treats functions as first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.
  + Higher Order Functions
    - Higher-order functions are functions that take other functions as arguments or return them as results.

**JS Scope**

* + Kinds of Scope
    - Global Scope: *Variables declared outside of any function or block have global scope.*
    - Local Scope: *Variables declared within a function have local scope.*
    - Block Scope: *Variables declared using let and const are scoped to the block (any pair of curly braces {})*
  + Lexical Scope
    - Scope Resolution Based on Source Code Structure:*Lexical scoping determines variable scope based on the structure of the source code, specifically where variables are defined and where they are being used, at the time the code is written (i.e., during the code's compilation or parsing phase). This contrasts with dynamic scoping, where scope can depend on the runtime flow of the program.*
    - Nested Functions Inherit Scope:*In a lexically scoped language, nested functions have access to variables from their containing (parent) function's scope. This is because the scope of a function is determined by where the function is defined in the source code, not where it's called or executed.*

**JS Engine**

* + JavaScript Engine
    - A JavaScript engine is a software component that interprets and executes JavaScript code.
    - It's responsible for taking the JavaScript code you write and translating it into instructions that the computer can understand and execute.
    - JavaScript engines are an integral part of web browsers, server-side environments, and other platforms where JavaScript is used.
  + Some of JS Engine
    - V8 (Used in Google Chrome and Node.js)
    - SpiderMonkey (Used in Mozilla Firefox)
    - JavaScriptCore (Used in Safari)
    - Chakra (Legacy, used in Microsoft Edge)

**JS: Interpret vs Compile**

* + JS Interpretation:Traditionally, JavaScript was primarily interpreted by web browsers. This means that the browser's JavaScript engine reads the source code line by line and executes it in real-time. Each time a script is loaded, the engine reads the code, interprets it, and executes the corresponding actions. Interpretation allows for flexibility and immediate execution but can lead to lower performance due to the repeated parsing and execution of the same code.
  + JS Compilation:Modern JavaScript engines, such as Google's V8, use a combination of interpretation and compilation to improve performance. These engines employ a technique called Just-In-Time (JIT) compilation.

**JS Execution Context**

* + Define
    - In JavaScript, an execution context is a fundamental concept that plays a crucial role in how code is executed. It provides an environment in which JavaScript code is evaluated and executed. Each time a function is invoked, a new execution context is created. Understanding execution contexts is essential for grasping concepts like variable scope, hoisting, and the "this" keyword.
    - JavaScript uses a call stack to manage execution contexts. The call stack is a data structure that keeps track of which execution context is currently running and manages the order in which functions are called and returned.
  + Kinds Of Execution Context
    - G**lobal Execution Context:**This is the default context when your code starts running. It represents the global scope and any code outside of a function resides here. Variables and functions declared in the global scope are accessible within this context.
    - **Function Execution Context:** Every time a function is called, a new function execution context is created. It includes information about the function's local variables, parameters, and the value of the "this" keyword within the function. When a function is done executing, its execution context is popped off the call stack.
    - **Eval Execution Context:** The "eval()" function in JavaScript can be used to execute code dynamically. It creates a new execution context where the provided code is evaluated. However, using "eval()" is generally discouraged due to security and performance concerns.
  + Execution Context Have Two Phase
    - **Creation Phase:** In this phase, the JavaScript engine sets up the context. It does things like creating the Variable Object (which holds function arguments, inner variable declarations, and function declarations), determining the value of "this", and establishing a scope chain (used for variable lookup).
    - **Execution Phase:** Once the creation phase is complete, the code is executed line by line. During this phase, variables are assigned their values, and functions are invoked.

**JS OOP**

* + OOP √√√
    - Object-Oriented Programming (OOP) is a programming paradigm that organizes and structures code around the concept of "objects," which are instances of classes.
    - It aims to model real-world entities, concepts, or systems as individual objects that interact with each other to achieve desired functionalities.
    - OOP promotes the principles of encapsulation, inheritance, and polymorphism to enhance code modularity, re-usability, and maintainability.
  + OOP in JS: √√√
    - Literals
    - Factory Function
    - Constructor Function
    - ES6 Class Syntax
  + Five(5) Pillars of OOP √√√
    - **Classes & Objects:** A class is a blueprint or template that defines the properties (attributes) and behaviors (methods) that an object of that class will possess. Objects are instances of classes, representing specific instances or items.
    - **Encapsulation:** Encapsulation refers to the practice of bundling data (attributes) and the methods (functions) that operate on the data into a single unit called an object. It provides access control mechanisms to ensure that data is only manipulated through defined methods, which helps maintain data integrity and control.
    - **Inheritance:** Inheritance allows a new class (subclass or derived class) to inherit properties and behaviors from an existing class (superclass or base class). This promotes code reuse and allows for the creation of hierarchies of related classes.
    - **Polymorphism:** Polymorphism enables objects of different classes to be treated as objects of a common base class. It allows the same method to exhibit different behaviors based on the type of object invoking it. This promotes flexibility and extensibility in code design.
    - **Abstraction:** Abstraction involves simplifying complex reality by modeling classes based on essential attributes and behaviors while hiding unnecessary implementation details. It helps programmers focus on relevant aspects of an object and its interactions.
  + OOP Keywords
    - **this:** *Refers to the current instance of a class.*
    - **constructor:** *Special method used for object initialization.*
      * A constructor is a special method that is used to initialize an object when it is created.
      * Constructors are typically used to set initial values for an object's attributes or properties and to perform any necessary setup or initialization tasks.
      * Constructors are called automatically when an object is instantiated (created) from a class.
    - **class:** *Used to define a new class.*

**JS Destructuring**

* + String Destructure
    - let [a,b,c,...rest] = “String”
  + Array Destructure
    - const [arr1,arr2,...arr] = [1,2,3,4,5,6,7,8]
  + Object Destructure
    - simple
      * const{uname, roll, age} = {uname: ‘Sushil’, roll: 24, age: 23}
    - different variable names while destructuring
      * const{uname: userName, roll: userRoll} = {uname: ‘Sushil’, roll: 24, age: 23}
    - Nested object destructuring
      * const{address: {street, city, country}, roll, uname: userName} = {address: {street: ‘Road-11’, city: ‘Rangamati’, country: ‘Bangladesh’}, roll: 23, uname: ‘Sushil’}
    - Default values can also be provided
      * const{uname = ‘Sushil’} = {uname: undefined}
    - Destructure within function parameters
      * function({uname, roll, age}){}

**JS Module**

* + Module System
    - The JavaScript module system is a way to organize and structure code in modular fashion, allowing developers to break down their codebase into smaller, manageable pieces. This improves code reusability, maintainability, and collaboration among developers. The module system was introduced as part of the ECMAScript 2015 (ES6) standard.
  + Kinds of Module System
    - **default export to import**
      * export default let uname = ‘Jagadish Chakma’
      * import myName from ‘./module/user.js’
      * default export only use one time on a single file
    - **default export with normal exports to import**
      * default export let name = ‘Jagadish Chakma’; export let age = 23, roll = 44
      * import myName, {age, roll} from ‘./modules/user.js’
    - **normal exports to import**
      * export let age = 23, roll = 44
      * import {age, roll} from ‘./modules/user.js’
    - **export to import change name**
      * default export let name = ‘Jagadish Chakma’; export let age = 23, roll = 44
      * import {default as uname, age as boyos, roll as classRoll} from ‘./modules/user.js’
    - **Re-exports**
      * export {default as uname, age as boyos, roll as classRoll} from ‘./modules/user.js’

**JS Error**

* + Definition
    - In JavaScript, errors occur when there's a problem in the code that prevents it from running correctly. When an error occurs, JavaScript generates an error object that contains information about the error, including its type and additional details. These error objects can be helpful in diagnosing and fixing issues in your code.
  + Kinds Of Error
    - Syntax Error
    - Reference Error
    - Type Error
    - Range Error
    - Internal Error
    - Custom Error
  + Error Global Object
    - new Error()
    - properties
      * name
      * message
  + throw statement
    - throw new Error(message)
  + try…..catch statement
    - try{}catch(error){}finally{}
  + Error Handling
    - Promise Rejection Handling
    - Async/await Error Handling
  + Custom Error Handling
    - try{throw new Error(‘Error occurred for you}catch(e){console.log(e.message)}

**JS Asynchronous**

* + Synchronous
    - Synchronous programming refers to a style of programming where tasks are executed one after the other, in a sequential and predictable order, ensuring that each task completes before the next one starts.
  + Asynchronous
    - Asynchronous programming involves executing tasks independently, allowing them to overlap and proceed concurrently, without waiting for each task to finish before starting the next one. This enables more efficient use of resources and is particularly useful for handling tasks that might take time, such as I/O operations, without causing the entire program to wait.
  + Promise
    - async/await

**JS Prototypes**

* + Prototype Definition √√√
    - Prototype is a any kind of functions property. That is itself a object.
  + Prototype Editable √√√
    - Only constructor function’s prototype editable
    - Other built-in prototype are not editable.
  + Prototypes Behavior √√√
    - In JavaScript, prototypes are a fundamental concept in the language's object-oriented programming model.
    - Prototypes allow you to define a "template" or a set of shared properties and methods that can be inherited by multiple objects.
    - This helps in creating a hierarchy of objects with shared behaviors.
  + Prototype Chain
    - Each object in JavaScript has a prototype, which can be another object.
    - Objects inherit properties and methods from their prototypes.
    - This inheritance forms a chain known as the "prototype chain."
  + Constructor Functions
    - Constructors are used to create objects with a shared prototype.
    - Constructors are just regular functions, but they are typically named with an initial uppercase letter (Person, Car).
    - The new keyword is used to instantiate objects from constructor functions.
  + Prototype Property
    - Each constructor function has a prototype property, which points to an object.
    - This prototype object is shared among all instances created from that constructor.
    - Methods and properties added to the prototype object are inherited by all instances.
  + Instance Properties and Methods
    - `Instance-specific properties are often defined inside the constructor using the this keyword.
    - Instance methods can also be defined inside the constructor, but this leads to redundant function creation for each instance.
  + Prototype Methods
    - To avoid redundancy, methods that should be shared among instances are usually added to the constructor's prototype.
    - These methods can be accessed by instances via the prototype chain.
  + Object.prototype
    - All objects, including functions, inherit from the Object.prototype.
    - Methods like toString() and hasOwnProperty() are defined on Object.prototype.
  + Creating Prototypal Relationships
    - Objects inherit from other objects through their prototypes.
    - You can establish this relationship using the Object.create() method or by directly setting the prototype property of the constructor function.
  + instanceof Operator
    - The instanceof operator checks if an object is an instance of a particular constructor.
    - It does this by traversing the prototype chain.
  + ES6 Class Syntax
    - ES6 introduced a class syntax that simplifies creating constructor functions and prototypes.
    - Under the hood, it still utilizes prototypes and prototype chains.
  + super Keyword
    - The super keyword is used in classes to call methods of the parent class's prototype.
    - It's used in constructor functions as well as in methods.

**JS ES6 or Later**

* + **Variables1**
    - let
      * not re-declare
      * re-assigned
      * block scope
    - const
      * not re-declare
      * not re-assigned
      * block scope
  + **String2**
    - Template Literals
      * String template are a feature in JavaScript that allow you to embed expressions and variables inside strings using a special syntax.
      * This makes string interpolation and multi line strings much more convenient compared to traditional string concatenation.
      * `back tick`
      * `${reference value}`
    - Destructuring String
      * You can destructure strings using the spread syntax to treat them as arrays of characters.
      * Let [a,b,...rest] = “string”
      * Let[a,b,c...rest] = [1,2,3,4,5]
    - Unicode Code Point Escapes
      * Represent Unicode characters using their code point escapes in string literals.
      * console.log(‘\u{1F60D}’)
    - String Methods
      * startsWith()
      * endsWith()
      * includes()
      * trimStart()
      * trimEnd()
      * padStart()
      * padEnd()
  + **Functions3**
    - Arrow Function
      * Arrow functions have a shorter syntax when compared to regular function expressions, and they can be especially handy for simple functions.
      * No Binding of ‘this’
      * No Arguments Object
      * No Constructor Usage: Arrow functions cannot be used as constructors. You cannot use the `new` keyword to create instances of an arrow function.
    - Default Parameter
      * Default parameters are a useful feature that simplifies handling optional arguments and can make your code more readable and maintainable.
  + **Operators4**
    - Spread Operator
      * Copying Arrays: You can use the spread operator to create a shallow copy of an array.
      * Merging Arrays: You can merge two or more arrays using the spread operator.
      * Passing Function Arguments: You can use the spread operator to pass an array as individual arguments to a function.
      * Creating Array Copies with Additional Elements: You can create a new array by adding elements to an existing array using the spread operator.
      * Converting String to Arrays: You can use the spread operator to convert a string into an array of characters.
      * Creating Object with Merged Properties: While the spread operator is primarily used with arrays, it can also be used to merge properties of objects.
      * ...(three dots)
      * The spread operator is a powerful tool for simplifying various operations in JavaScript and is commonly used to improve code readability and maintainability.
    - Rest Operator
      * Collecting Remaining Arguments:
      * Destructure Remaining Elements:
      * Characteristics:
        + Position in Parameter List: The rest parameter should be the last parameter in the function's parameter list, as it collects all remaining arguments. You can't have other parameters after the rest parameter.
        + Accessing Arguments: The rest parameter is treated as an array, so you can access its elements using array indexing and iteration methods.
        + No Arguments Object: When using the rest operator, the arguments object (which is available in regular function definitions) is not accessible within the function.
      * ...(three dots)
      * The rest operator is especially useful when you want to create flexible functions that can handle a variable number of arguments while maintaining code readability and avoiding the use of the deprecated arguments object.
  + **Objects5**
    - Enhance Object
      * shorthand property name: {x, y}
      * shorthand property method: {print(){}}
    - Destructuring Object
    - Computed Property Names
      * {[varName]: value}
    - OOP Object-Oriented
      * class keyword
      * properties & methods define
      * static keyword
      * this keyword
      * private property
  + **Data Types6**
    - Symbol
      * Think of a symbol as a special key that you can use to add properties to objects without worrying about name collisions with other properties. Symbols are often used as property keys to ensure uniqueness.
  + **Loops7**
    - for...of loop
    - for…in loop
  + **Module8**
    - whole package
  + **Error Handling9**
    - try{}catch{}finally{}
    - for optimized,memory efficient, speed up