Day7 + Day8

Responsive Web Design from Figma design

a. Figma link

Day9

Js

Var, let, const

a. Var

i. Var is of function scope and is hosted

```
function f() {

    // It can be accessible any
    // where within this function
    var a = 10;
    console.log(a)
}
f();

// A cannot be accessible
// outside of function
console.log(a);
```

ii. Can be redeclared

```
var a = 10

// User can re-declare
// variable using var
var a = 8

// User can update var variable
a = 7
console.log(a);
```

iii. There is hoisting

1. According to concept of execution context, first of all variables and function are first declared, (variables are initialized to undefined, and functions are assigned their own execution context)
2. Now after all variables and functions are declared to undefined, then only, the code is interpreted line by line to assign real values and replace the undefined stuff.
3. So, if we var a = 10 in line2, but console.log(a) in the first line, then still the first line will print undefined, even when a was not declared yet.

```
console.log(a);
var a = 10;
```

b. Let

i. Introduced in ES6 (2015)

ii. Block scope

```
if (true) {
    let b = 9

    // It prints 9
    console.log(b);
}

// It gives error as it
// defined in if block
console.log(b);
```

iii. Cannot be redeclared

```
let a = 10

// It is not allowed
let a = 10

// It is allowed
a = 10
```

iv. Hoisted but in **temporal dead zone** until initialization (Uncaught ReferenceError: Cannot access 'a' before initialization)

```
console.log(a);
let a = 10;
```

c. Const

i. has all the properties that are the same as the let keyword, except the user cannot update it and have to assign it with a value at the time of declaration.

ii. But using const to declare objects, the keys of objects cannot be changed, but values of those keys can be updated.

```
const a = {
    prop1: 10,
    prop2: 9
}

// It is allowed
a.prop1 = 3

// It is not allowed
a = {
    b: 10,
    prop2: 9
}
```

d. Difference between var, let, const

var	let	const
The scope of a *var* variable is functional or global scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared in the same scope.	It can be updated but cannot be re-declared in the same scope.	It can neither be updated or re-declared in any scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
These variables are hoisted.	These variables are hoisted but stay in the temporal dead zone untill the initialization.	These variables are hoisted but stays in the temporal dead zone until the initialization.

e. **Hoisting**
    i. **Hoisting** simply gives higher specificity to JavaScript declarations. Thus, it makes the computer read and process declarations first before analyzing any other code in a program.
    ii. A **temporal dead zone** (TDZ) is the area of a block where a variable is inaccessible until the moment the computer completely initializes it with a value.
        1. a let (or const) variable's TDZ ends when JavaScript fully initializes it with the value specified during its declaration.
        2. a var variable's TDZ ends immediately after its hoisting—not when the variable gets fully initialized with the value specified during its declaration.

```
let bestFood // 1. JavaScript parsed the first bestFood declaration

let myBestMeal // 2. the computer parsed myBestMeal variable declaration

bestFood = "Fish and Chips"; // 3. the computer initialized the bestFood variable

myBestMeal = function () {
  console.log(bestFood);
  let bestFood = "Vegetable Fried Rice";
}; // 4. JavaScript initialized myBestMeal variable

myBestMeal(); // 5. the computer invoked myBestMeal's function

let bestFood // 6. JavaScript parsed the function's bestFood declaration

console.log(bestFood); // 7. the computer parsed the function's console.log statement

Uncaught ReferenceError // bestFood's invocation returned an Error
```

So, what's happening here?

a. First of all, bestFood was declared and assigned undefined
b. Then myBestMeal was declared and assigned undefined
c. Line3 is skipped since it is already declared and assigned undefined
d. Now, myBestMeal func is declared.
e. Now, that all func are declared, lets move to myBestMeal
f. Here block scope is followed,

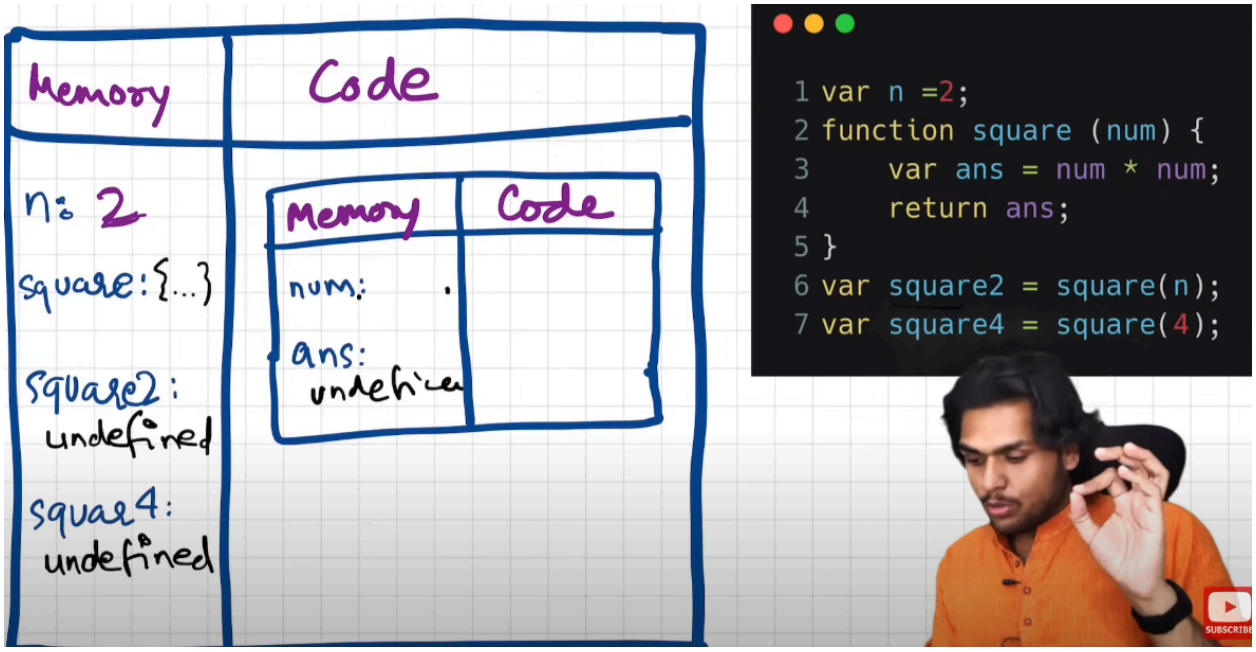
g. JavaScript parsed the function's bestFood declaration.
h. The computer parsed the function's console.log statement, when bestFood is not even assigned yet
i. So, reference error : a is not initialized yet is printed out.

```
// first of all a is declared and assigned to undefined
// then on the block scope, console log is skipped for declaration phase
// now again a is declared and undefined for block scope
// now line by line execution starts
// so, for block scope, when a is undefined, and is in temporal scope, console log will print reference error

let a = 10;

if (true) {
    console.log(a);
    let a = 9;
}
```

Execution Context and Call Stack



Datatypes

Data Type	Description	Example
`String`	Textual data.	`'hello'`, `"hello world!"`, etc.
`Number`	An integer or a floating-point number.	`3`, `3.234`, `3e-2`, etc.
`BigInt`	An integer with arbitrary precision.	`900719925124740999n`, `1n`, etc.
`Boolean`	Any of two values: `true` or `false`.	`true` and `false`
`undefined`	A data type whose variable is not initialized.	`let a;`
`null`	Denotes a `null` value.	`let a = null;`
`Symbol`	A data type whose instances are unique and immutable.	`let value = Symbol('hello');`
`Object`	Key-value pairs of collection of data.	`let student = {name: "John"};`

Primitive vs Non Primitive datatypes

```
//primitive = {STRING, NUMBER, BOOLEAN, SYMBBOL, UNDEFIEND, NULL}
//non-primitive = {ARRAY, OBJECT, FUNCTION}

//primitive is always copied by VALUE
var a = 1;
var b = a;
    //console.log(a , b) =  1 , 1
a = 3;
console.log(a) //3
console.log(b) // still 1 and not 3   (always copied by value only)


//non-primitive is always copied by REFERENCE
var x = {name : "Jscript"};
var y = x;
  //console.log(x , y)     TWICE =  Object { name: "Jscript" }
x.name = "Js";
console.log(x) //Js
console.log(y) //Js  {copied by reference} like pointers in C lang
```

Primitive

a. Number
   i. Both decimal and non decimal hold
   ii. can safely store and operate on large integers even beyond the safe integer limit (Number.MAX_SAFE_INTEGER) for Numbers.

```
let x = 250;
let y = 40.5;
console.log("Value of x=" + x);
console.log("Value of y=" + y);
```

b. String
   i. sequence of characters that are surrounded by single or double quotes.
c. Undefined
   i. has been declared but has not been assigned a value, or it has been explicitly set to the value `undefined`.
d. Boolean
   i. boolean data type can accept only two values i.e. true and false.
e. Null
   i. This data type can hold only one possible value that is null.

```
let x = null;
    console.log("Value of x=" + x);
```

f. BigInt
    i. BigInt data type can represent numbers greater than 253-1 which helps to perform operations on large numbers. The number is specified by writing ‘n’ at the end of the value
g. Symbol
    i. create objects which will always be unique and immutable.
    ii. these objects can be created using Symbol constructor.

```
let sym = Symbol("Hello")
console.log(typeof(sym));
console.log(sym);
```

    iii. Symbols can be used as constants

```
const COLOR_RED = Symbol('red');
const COLOR_GREEN = Symbol('green');

function getColorName(color) {
  switch (color) {
    case COLOR_RED:
      return 'Red';
    case COLOR_GREEN:
      return 'Green';
    default:
      return 'Unknown color';
  }
}
```

    iv. Symbols can be used for unique property keys

```
const uniqueKey = Symbol('uniqueKey');

const obj = {
  [uniqueKey]: 'This is a unique value',
  commonKey: 'This is a common value'
};

console.log(obj[uniqueKey]); // Output: 'This is a unique value'
```

[General Knowledge]

a. NaN

i. special kind of number value that's typically encountered when the result of an arithmetic operation cannot be expressed as a number.
ii. is the only value in JavaScript that is not equal to itself.

Non Primitive

a. Object
   i. a collection of properties
   ii. a limited set of properties are initialized; then properties can be added and removed
   iii. Object properties are equivalent to key-value pairs.
   iv. **Keys** can be strings or symbols.
   v. **Properties** of objects can be broadly classified into two categories: data properties and accessor properties.
   vi. A **data property** is a property that directly contains a value. Data properties have four attributes:
      1. Value: The actual value stored in the property.
      2. Writable: A boolean indicating whether the value of the property can be changed.
      3. Enumerable: A boolean indicating whether the property can be enumerated in a for...in loop or Object.keys.
      4. Configurable: A boolean indicating whether the property can be deleted or changed (excluding the value).

```
Object.defineProperty(obj, 'name', {
    value: 'Suyan',
    writable: true,
    enumerable: true,
    configurable: true
  });

  console.log(obj.name); // Output: Suyan
```

   vii. An **accessor property** does not contain a value directly. Instead, it defines a pair of functions: a getter and a setter. These functions are called when the property is accessed or modified, respectively.
      1. Get: A function called when the property is read.
      2. Set: A function called when the property is written to.
      3. Enumerable: A boolean indicating whether the property can be enumerated in a for...in loop or Object.keys.
      4. Configurable: A boolean indicating whether the property can be deleted or changed (excluding the getter and setter functions).

   viii. When configurable is true,

1. you can change the other attributes of the property (writable, enumerable, value, get, set).
2. the property can be deleted from the object using the delete operator.

```
// Creating objects
// --------------
// using constructor functions
// Create an empty generic object
let obj = new Object();

// Create a user defined object
let mycar = new Car();
// --------------
// using literal notations
// An empty object
let square = {};

// Here a and b are keys and
// 20 and 30 are values
let circle = {a: 20, b: 30};
```

Operators

a. == Equal to
b. === Strictly equal to
c. != Not equal to
d. !== Strictly not equal to
e. > Greater than
f. < Less than
g. >= Greater than or equal to
h. <= Less than or equal to

JSON

a. JSON is a text-based data format following JavaScript object syntax.
b. Keys must be strings and enclosed in double quotes. (not single quotes, strictly double quotes)
c. Values can be of any data type (string, number, object, array, boolean, null).
d. JSON is more strict than object