# Lec 2: Function

1. We can acces the properties of objects like: obj[“property\_name”]

Ex:

*const* alok = {

  firstName: "Alok Ranjan",

  lastName: "Joshi",

  age: 2023 - 2002,

};

console.log("firstName", alok["firstName"]);

It’s useful in selecting any property dynamically.

For example:

*const* nameKey = "Name";

console.log(alok["first" + nameKey]);

console.log(alok["last" + nameKey]);

# Lec 8: Behind the Scenes

## Priority of variables having same name in different scope

* In Javascript, It first check the variable in the current scope. If not found then it'll check in the parent scope.
* Let an variable having some name is declared in the parent scope using const keyword. Then also we can declare same variable using const keyword in it's child scope. The value of the variable in the child scope will have more priority than that of the parent scope if it's used inside the child scope.

#### Creation of variable that are declared with different keywords.

* The variable declared with **var** keyword are executed first during the execution of the js file but the values are not initialized to them, I.e. the variable are set to undefined. The variables are initialized during the flow of execution of code.
* But the variable declared with **let**, **const** keyword are not created first. They are created and initialized in the flow of execution of code.

#### Scope of regular function

* The functions are normally **block level** scope. But when the **“strict-mode”** is not specified, they are **function scoped**.

#### TDZ (Temporal Dead Zone)

* The lines that are present before the declaration of variable using **let** or **const** keyword are the **TDZ** of that variable.

#### Window object and var keyword:

* Variables created using “**var**” keyword will be a member of **window** object. We can access them using **window.variable\_name** also.

#### “this” keyword and different types of function

* The arrow function **inherits** the “this” keyword of it’s **parent function** otherwise it give the window object.
* The anonymous function represent the object in which it is defined otherwise it give **undefined**.

#### “argument” object and different types of function

* Arrow function doesn’t contains the **arguments** object whereas the regular and anonymous function contains.
* ***NOTE:*** Let a function have only 2 arguments are specified in it’s declaration and if we call the function with more than 2 arguments then we can access all the arguments using the arguments object.

#### Primitive and Reference types *(Most important*):

*// Primitive Types*

*let* x = 5;

*let* y = x;

x = 2;

console.log(x, y); *// 2 5*

*// Reference Types*

*const* mee = {

  name: 'Alok',

  age: 21,

};

*const* friend = mee;

friend.age = 27;

console.log('mee', mee); *// Alok, 27*

console.log('friend', friend); *// Alok, 27*

* There are two memory pools present I.e. **call stack**, and **heap.**
* The values in a perticular address in call stack is immutable but in heap it is mutable.
* All the primitive variables are stored in call stack where as the function, arrays, objects literals are stored in the heap.
* ***NORMAL VARIABLE:***
* When we declare a normal variable, the value assigned to it will be stored in a block in the call stack and the address of that block is stored in the variable. In the above example, the address in which value 5 is stored will be stored in x. **SImply x will point to the block in which 5 is stored.**
* Now when we assign x to y, y also points to the same block which stores 5.
* Now when we try to change the value of x = 2, **An another block will be storing the value 2 and x now point to that block.**
* So, here no side effect will be there.
* ***OBJECT LITERALS:***
* When we create a object literal (here mee), It'll be stored in the heap (let H1). now H1 will be stored in call stack(let CS1) to which the variable mee will point. It means "mee" will now point to CS1, which contains the address H1 in it. And h1 contains the actual values i.e. name: Alok, age: 21. **Simply, mee points to a block in the call stack that points to the block in the heap which stores the data.**
* Now when we assign the mee object to another object (here friend), the friend variable now refer to the same location i.e. CS1. Now when we try to change the value inside the "mee" object, it'll change in the heap.

# Lec 9: Data Structures Operators

#### Destructuring Array:

*const* arr = [4, 2, 7, 9, 1, 5, 3, 8];

*let* [a, b, , d] = arr;

console.log(a, b, d); *// 4 2 9*

Here as we’ve given a extra comma, so the value at index-2 I.e. 7 is skipped.

*// Swapping two values*

console.log(a, b); *// 4 2*

[a, b] = [b, a];

console.log(a, b); *// 2 4*

#### Destructuring Object

If the same property name will have to be kept:

*const* { name, starterMenu, categories } = restaurant;

If the property name have to be changed:

*const* {

  name: restaurentName,

  starterMenu: restaurentMenu,

  categories: tags,

} = restaurant;

If any default value has to be set to any property if it is not present inside the object which has to be destructured:

*const* { menu = [], starterMenu: starters = [] } = restaurant;

Mutating variables:

*let* m = 456;

*let* n = 221;

*const* obj = { m: 34, n: 56 };

*// {m, n} = obj; // It will be error as {} is understood as a block by Javascript.*

({ m, n } = obj);

#### Spread Operator(…)

It simply write all the values present in side the array being separated by a comma (,)

*IMPORTANT:*

*\*: A very big difference between array destructuring and sprear operator(...) is that spread operator doesn't create new variables like destructuring and the values can be used only when it is needed.*

*NOTE: The spread operator can be used in all the iterables like arrays, strings, maps, sets. Objects are not iterable but spread operator works on objects too.*

***// Use in function***

***function* orderPasta(*ing1*, *ing2*, *ing3*) {**

**console.log(`ordered paste containing ${ing1}, ${ing2}, and ${ing3}.`);**

**}**

***const* ingredients = ['a', 'b', 'c'];**

**orderPasta(...ingredients);**

#### Rest Pattern and Parameters

***const* anArray = [1, 2, ...[3, 4]]; *// spread because on RIGHT side of =***

***const* [first, second, ...others] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]; *//REST,  because on LEFT side of =***

***// Functions***

***const* add = *function* (...*values*) {**

**console.log(values); *// [4, 3, 5, 3, 6, 3, 5, 7]***

**};**

**add(3, 5);**

**add(4, 3, 5, 3, 6, 3, 5, 7); *// It simply like add(...[4, 3, 5, 3, 6, 3, 5, 7])***

#### Short Circuiting && ||

*In case of || The fist truthy operand will be executed. If all the operands are falsy then the last falsy operand will be executed.*

**console.log(3 || 'Alok'); *// 3***

**console.log('' || 'Alok'); *// Alok***

**console.log(true || 0); *// true***

**console.log(undefined || null); *// null***

**console.log(undefined || 0 || '' || 'Hello' || 23 || null); *// Hello***

*In case of && the first falsy operand will be executed. If all are true then the last truthy operand will be executed.*

**console.log(0 && 'Alok'); *// 0***

**console.log(7 && 'Alok'); *// Alok***

**console.log('Hello' && 23 && null && 'Alok'); *// null***

#### Nullish Coalescing Operator ??

#### Logical Assignment Operators