CLASSTEAM° | Page No.
Date / /

JS - Basics - II

OBJECTS in JavaScript

→ Objects may be defined as an unordered collection of related data, of primitive or reference types, in the form of "key: value" pairs within {...}.

→ These objects are quite different from JavaScript's primitive data type (Number, String, Boolean, Null, Undefined).

* Object Creation :-

→ USING OBJECT LITERALS —

(1.)
```
let Rectangle = {
    length : 1,
    breadth : 2,
    draw : function () {
        console.log ("Drawing Rectangle");
    }
}
```

Rectangle → Variable
length, breadth → property / key
1, 2 → Value
draw → method

Note:- Here, we create js objects with object literal.

* //property accessor DOT Notation

output :-

```
console.log (Rectangle.length);        → 1
console.log (Rectangle.breadth);       → 2
Rectangle.draw();                      → Drawing Rectangle
```

* // Bracket Notation with single quote '' or ""

```
console.log (Rectangle['length']);     → 1
console.log (Rectangle['breadth']);    → 2
```

* Adding NEW property to the object

ex:-

```
let car = {
        name: 'GiT',
        maker: 'BMW',
        engine: '1998 cc'
    };

//adding NEW property to the object here
car.brakesType = 'All Disc';

console.log (car);
```

Output:- { name: 'GiT', maker: 'BMW', engine: '1998cc', brakesType: 'All Disc' }

* Adding NEW methods to the objects

ex:-

```
let car = {
        name: 'GiT',
        start: function (){
            console.log ('Starting Engine');
        }
    };
```

Adding New methods like ↓

```
car.stop = function (){
        console.log('Applying Brake');
    }
```

```
car.start();      //Calling Method Access
car.stop();       //Accessing Calling method
```

Output:-

Starting Engine.
Applying Brake

To Create Object of Same type again is Bulky code.

So, we use two Object Creation functions –

(i) Factory function. (CamelCase Notation)
(ii) Constructor function. (Pascal Notation)

→ **Factory Function** (camelCase Notation)

Factory Function in JS are similar to constructor function, but they do not require the use of 'this' keyword for inner values OR the use of 'new' keyword when instantiating new objects.

Factory Functions differ from Regular functions as they always return an object, which will contain any value, method etc.

Why it is useful?

If we have complex logic and we have to create multiple objects again and again that have the same logic.
Then, we create a function to write logic once as a factory to create our products.

Example :- //factory function (Return value को किसी variable में store भी कराना पड़ेगा)

```
function createRectangle() {
    let rectangle = {
        length: 1,
        breadth: 2,
        draw: function() {
            console.log('drawing rectangle');
        }
    };
    return rectangle;
}
let a = createRectangle();   // Object Creation. (Objects Return करेगा)
console.log(a);
```

② In a Simple return & Simple draw() (we can Simplify)

```
function createRectangle() {
    return rectangle = {
        length: 1,
        breadth: 2,
        draw() {
            console.log('drawing rectangle');
        }
    };
}
```

we can create like draw()

~~rectangle.draw();~~ ~~//Avoiding getting draw()~~

Returned object, a లో store అవుతుంది

```
let a = createRectangle();   (// store into a)
console.log(a);
```

→ {length: 1, breadth: 2, draw: f}

```
a.draw();
```

→ Drawing Rectangle.

Simple Representation Conclusion:-

Function Call → Request Object creation → Factory Function

Return Object using (Return keyword)

Dynamic Nature

Adding Properties and Methods (which is not in objects)

① a.Extra = 'width';

② a.erase = function() { console.log("Erasing Rectangle"); }

console.log(a); → {length: 1, breadth: 2, Extra: 'width', draw: f, erase: f}

Deleting Properties

→ delete a.Extra; ✓ (Extra: 'width') ✗

→ delet a.draw; draw() ✗ → It will be deleted from a object

ex ③ Factory Function with Input Parameters.

```
function createRectangle(len, bre) {
    return rectangle = {
        length: len,
        breadth: bre,
        draw() {
            console.log('drawing rectangle');
        }
    };
}

let rectangleObj1 = createRectangle(5,4);
console.log(rectangleObj1);
```

We can use only { len, bre, draw() { } }

Output:- {length: 5, breadth: 4, draw: f}

Note:- We can directly use 'length' & 'breadth' or any Input Name as property without Key: Value pair.

Ex:-

```
function createRectangle(length, breadth) {
    return rectangle = {
        length,
        breadth,
        ------
    }
}
```

CamelCase Notation :- first letter is small of 1st word and first letter of other word is Capital.

→ Constructor Function (Pascal Notation)

Pascal Notation :- First letter of every word is Capital.
ex:- NumberOfStudent.

* Constructor function is a function which ~~initializes~~ initialize & defines the properties and methods of an object. by using this keyword.

example:-

```
function Rectangle() {
    this.length = 1;          //define length=1
    this.breadth = 2;         //define breadth=1
    this.draw = function() {
        console.log('drawing');
    }
}
```

this = empty object का show का रहा है।

* Note :- "this" keyword refers to showing Current Object.

* Empty object → obj = { };

No Need to Return in Constructor function

* "new" keyword is return an empty object.

example (1.)

```
function Rectangle() {
    this.length = 1;
    this.breadth = 2;
    this.draw = function() {
        console.log('Drawing Rectangle');
    };
}

let rectangleObject = new Rectangle();

// Object Creation using Constructor function.

console.log(rectangleObject);
rectangleObject.draw();
```

this = rectangleObject. का show का रहा है।

Output :-

```
Rectangle {length: 1, breadth: 2, draw: f}
Drawing Rectangle
```

* Constructor function with Input Parameters

```
function Rectangle (len, bre){
    this.length = len;
    this.breadth = bre;
    this.draw = function(){
        console.log("Drawing Rectangle");
    };
}

let rectangleObject = new Rectangle(4,6);
console.log(rectangleObject);
```

Output:-

```
Rectangle {length: 4, breadth: 6, draw: f}
```

Dynamic Nature of Objects [We discussed before]

→ Adding properties & methods to Objects.
→ Deleting properties.

Constructor Property :-

The constructor property returns the function that created the Object instance or prototype.

Note:- The value of this property is a reference to the function itself, not a string containing the function's name.

Syntax → `object.constructor`

example:-

```
console.log(rectangleObject.constructor);
```

↓ It will return a function value

Output:-

```
f Rectangle(len, bre){
    this.length = len;
    this.breadth = bre;
    this.draw = function(){
        console.log("Drawing Rectangle");
    };
}
```

Note:- `let rectangleObj = new Rectangle();`

(यहाँ construction के लिए function)

↓

अब यहाँ constructor क्या है?

P.T.O ↪

Object → Rectangle() → 0 ... constructor → constructor

* console.log (Rectangle. constructor);

Output:- Short Constructor, & default function.

```
f Function()
  {
    [native code]
  }
```

Note :- Function() is also an Object in JS.

→ Working :-

```
let Rectangle = new Function {
        'length', 'breadth',
        'this.length = length;
        this.breadth = breadth;
        this.draw = function(){
            console.log('Drawing Rectangle');
        };'
    }

let obj = new Rectangle (2, 3);
```

Back-tick Characters

Main

Difference :- Primitive Type Vs Reference Type.

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

a → 10, b → 10

Output:-
a → 11
b → 10

In Primitive Type,

* Copy of Value is created.

a → 10
b → 10

a++

a 11 ✓
b 10 ✓

→ Primitive are copied by their values.

```
let a = {value: 10};
let b = a;
a.value++;
console.log(a.value);
console.log(b.value);
```

Output :- a → 11
b → 11

In Reference Type,

* Copy is NOT Created.
* Point to Same address.

b → 10 → 11 obj ✓
a

→ References are Copied by their addresses / References.

Pass by Value

Example → Primitive Case

```
let a = 10;
increment(a);
function increment(a){
  a++;
}
increment(a);
console.log(a);
```

a → 10 (value passed)

Different Block: a 10, a++ 11

Outside Block: a 10

Output:- 10

Pass by Reference

Reference Case

```
let a = {value: 10};
incr(a);
function incr(a){
  a.value++;
}
incr(a);
console.log(a);
```

address passed

obj: 10 → 11

Output:- 11

Iterating through Objects.

① FOR-IN Loop

```
let rectangle = {
    length: 2,
    breadth: 4
}

for (let key in rectangle)        // key is Variable. Or i, j.
{
  // Keys are traversed through KEY Variable
  Console.log(key);
}
```

Output:- length, breadth

```
for (let key in rectangle)
{
  // Values are traversed through rectangle[key]
  console.log(rectangle[key]);
}
```

Output:- 2, 4

```
for (let key in rectangle)
{
  console.log(key, rectangle[key]);
}
```

key value pair

Output:- length 2, breadth 4

(2) FOR-OF Loop :-

This loop is applied to **Iterables**

- Array
- Maps
- Strings

[We will study further] →

Now,

Ques:- Why FOR-OF loop is applied directly to the object { } ?

Why Because object is not iterable.

Ex:-
```
let rectangle = {
                  length : 2,
                  breadth : 4
                }

for (let key of rectangle)
{
  console.log (key);
}
```

Output:-
ERROR:- 'Rectangle' is not iterable.

NOTE:-

* To Walk of FOR-OF loop using on Object

To make ~~Arrays of objects~~ variable using Object.keys in FOR OF loop.

example:-
```
let rectangle = {
                  length : 2,
                  breadth : 4
                }
```

(i)
```
for (let key of Object.keys(rectangle))
{
  console.log (key);
}
```

Output:- length ✓
breadth ✓

(ii)
```
for (let key of Object.entries(rectangle))
{
  console.log (key);
}
```

Output:- ['length', 2] ✓
['breadth', 4] ✓

To Check a property is exist or Not?

for example:-

```
let rectangle = {
                length : 2,
                breadth : 4
              }

if ('length' in rectangle ){
        console.log ('Present');
  }
else
  {
    console.log ('Absent');
  }
```

Output:-

Present

Object Cloning:-

(i) Iteration Cloning.
(ii) Assign Cloning (assign())
(iii) Spread Cloning. (...)

Cloning is defined as making a copy of the original object with same properties included it.

(a) Iteration Cloning:-

Syntax:-

```
let src = {key : value , key : value ------ };

let dest = { };

for (let key in src)
  {
    dest [key] = src[key];
  }

console.log (dest);
```

example of Iteration Cloning :-

```
let src = {
            a:10,
            b:20,
            c:30,
          };

let dest = {};          //empty object.

for (let key in src)
{
    dest[key] = src[key];
}

console.log (dest);

//to check cloning

src.att;
console.log (dest);     //successful cloning
```

(b.) Assign Cloning :-

Syntax:-

```
let dest = Object.assign({}, src);
```

example of Assign Cloning :-

```
let src = {
            a:10,
            b:20,
            c:30
          };

let dest = Object.assign({}, src);

console.log (dest);

//to check Cloning
  src.att;
  console.log (dest);       //successful cloning.
```

We can add more src{} to Object.assign.

({}, src, src2)

(c.) Spread Cloning :- Syntax:- {...src}

```
let src = {
            a:10,
            b:20,
            c:30
          };

let dest = {...src};

console.log (dest);         //successful cloning.

src.att;
console.log (dest);
```

Garbage Collection :-

Some high-level languages, such as JavaScript, utilize a form of automatic memory management known as Garbage Collection.

The purpose of garbage collector is to monitor memory allocation and determine when a block of allocated memory is no longer needed and reclaim it.

we have no control over G.C. and always Run in background.