

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 = {
 ↓ ↓ ↓ ↓ ↓
 type Variable Property length : 1, Value
 key breadth : 2,
 method draw : function () {
 } console.log ("Drawing Rectangle");
 } }

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

* Property accessor DOT Notation

```
console.log(Rectangle.length);
console.log(Rectangle.breadth);
console.log(Rectangle.draw());
Rectangle.draw();
```

Output :-

→ 1
→ 2
→ 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: 'GTR',
make: 'BMW',
engine: '1998 cc'

?;

Adding NEW property to the object here

car.brakesType = 'All Disc';

console.log(car);

Output:-

{ name: 'GTR', make: 'BMW', engine: '1998 cc',
brakesType: 'All Disc' }

* Adding Methods to the objects

Ex:-

let car = {

name: 'GTR',
start: function() {
console.log('Starting Engine');

?;

Adding New methods like ↴

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

~~Car.start();~~

car.stop();

^{Even}
Calling Method.
Actual Calling method

Output:-

Starting Engine.
Applying Brake.

To Create Object of same type again is ~~bulky~~
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. (Returning value in form of
object) // Store in variable

② `function createRectangle() {`

`let rectangle = {`

`length: 1,`

`breadth: 2,`

`draw: function () {`

`console.log('drawing rectangle');`

`}`

`} return rectangle;`

`let a = createRectangle();` // Object creation

`console.log(a);`

we can
Simplify

② In a simple return & simple draw() (Simplify)

`function createRectangle() {`

`return rectangle = {`

`length: 1,`

`breadth: 2,`

`draw() {`

`console.log('drawing rectangle');`

`}`

`rectangle.draw();` // Having nothing draw()

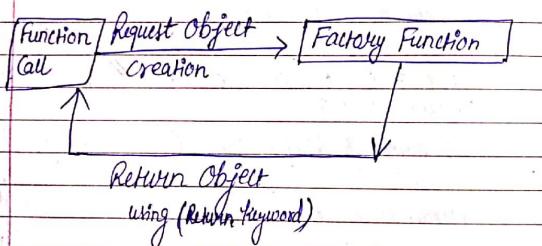
CLASSTEAM™ Page No. _____
Date _____

Returned object, a \Rightarrow store it in a)
 \downarrow

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

$\downarrow \{ \text{length: } 1, \text{breadth: } 2, \text{draw: } f \}$
 \downarrow
 $a.\text{draw}();$
 \downarrow
 Drawing Rectangle.

Simple Representation - Conclusion :-



Dynamically
Nature #

Adding Properties and Methods (which is not in objects)

- ① $a.\text{Extra} = \text{'width'}$;
- ② $a.\text{erase} = \text{function() \{ console.log("Erasing rectangle"); \}}$
 console.log(a); $\{ \text{length: } 1, \text{breadth: } 2, \text{Extra: } \text{'width'},$
 $\text{draw: } f, \text{erase: } g \}$

Deleting Properties

$\rightarrow \text{delete a.Extra}; \checkmark$ $\text{Extra: width} \times$
 $\rightarrow \text{delete a.draw};$ $\text{draw}() \times$
 $\downarrow \text{It will be deleted}$
 $\downarrow \text{from a object}$

Factory Function with Input Parameters.

function createRectangle(len, bre) {

return rectangle = {

length: len, \checkmark ,
 breadth: bre, \checkmark ,

We can use only
 { len, \checkmark ,
 breadth, \checkmark

draw() {

console.log('Drawing rectangle');
 } \checkmark

let rectangleObj1 = createRectangle(5, 4);

console.log(rectangleObj1);

Output:-

$\{ \text{length: } 5, \text{breadth: } 4, \text{draw: } f \}$

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, \dots

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

CLASSTEAM Page No.
Date / /

→ 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');  
    }  
}
```

3rd This
empty Object of
rectangle
on show on 2nd E1

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

* Empty object → obj = {} ;

No Need to Return in Constructor function

CLASSTEAM Page No.
Date / /

* "new" keyword is return an empty object.

example (1)

```
function Rectangle () {
```

```
    this.length = 1;  
    this.breadth = 2;  
    this.draw = function () {
```

```
        console.log('Drawing Rectangle');  
    }  
}
```

3rd This
rectangleObject
on show on 2nd E1

Let rectangleObject = new Rectangle();

// Object Creation using Constructor function.

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

Output :-

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

* Constructor function with Input Parameter

```
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: [Function]}
```

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);
```

Output:-

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

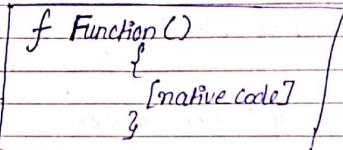
Note:- let rectangleObj = new Rectangle();
At construction is it E function
↓
At construction is it E?
↓
P.T.O

object → constructor → constructor

CLASSTEAM™ Page No.
Date / /

* `console.log(Rectangle.constructor);`

Output:- `function Constructor() { ... }` default function



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
Character

Main Difference :- Primitive Type Vs Reference Type.

```

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

Output:-
a → [11]
b → [10]

```

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

Output :- a → [11]
b → [11]

In Primitive Type,

* Copy of Value is Created.

```

a → [10]
b → [10]
a++;
a → [11] ✓
b → [10] ✓
  
```

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

```

a → [10]
b → [10] obj → [11] ✓
a → [11] ✓
  
```

→ Primitive are copied by their values.

→ References are copied by their addresses / references.

Pass by Value

Example → Primitive Data

```
let a = 10;
increment(a);
{
    a++;
}
increment(a);
console.log(a); //outside block
a
```

Different Block
a [10]

Output:- [11]

Pass by Reference

CLASSTEAM™ Page No.
Date / /

(8) FOR-OF Loop :-

This loop is applied to Iterables

```

    ↓      ↓      ↓
  [Array] [Maps] [String]
  
```

[We will study further] ↗

Now,

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

Ans: 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 Hack of FOR-OF loop using on Object variables
To make Arrays of objects 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 some properties included it.

(i) 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.

(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.