**5. Object**

**1) Object**:

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method. In addition to objects that are predefined in the browser, you can define your own objects.

**Example**:

let circle = {

radious: 1,

location: {

x: 1,

y: 2

},

isVisible: true,

draw: function() {

console.log("draw");

}

};

circle.draw(); *//draw*

**2) Factory Function**:

A factory function is any function which is not a class or constructor that returns a (presumably new) object. In JavaScript, any function can return an object. When it does so without the new keyword, it’s a factory function.

Factory functions have always been attractive in JavaScript because they offer the ability to easily produce object instances without diving into the complexities of classes and the new keyword.

**Example**:

*//factory function*

function createCircle(radious) {

return{

*/\**

*in modern JavaScript if the key and value are the same*

*then we can remove the key and just add the value*

*\*/*

radious,

draw(){

console.log("draw");

}

};

}

const circle1 = createCircle(10);

console.log(circle1); *//Object { radious: 10, draw: draw() }*

const circle2 = createCircle(20);

console.log(circle2); *//Object { radious: 20, draw: draw() }*

**3) Constructor Function**:

Constructor functions are the equivalent of classes in many programming languages. Sometimes people will refer to them as as reference types, classes, data types, or simply constructors. If you aren’t familiar with classes, they are a construct that allows you to specify some properties and behaviors (functions), and multiple objects can be created with those properties and behaviors. A common analogy you’ll often hear is, a class is to a blueprint as an object is to a house. Multiple houses can be created from a single blueprint, as multiple objects can be created from a class.

The job of Constructor function is to construct or create object. For use constructor function we should use Pascal notation (First letter should be upper case. Example: OneTwoThree.

**Example**:

function Circle(radius) {

*this*.radius = radius;

*this*.draw = function() {

console.log("draw");

};

}

const circle = new Circle(30);

console.log(circle); *//{ radius: 30, draw: draw() }*

When we use "new" operator 3 thing happening

1. This "new" operator create an empty JavaScript object like Circle ={};
2. Next it will set this(like radius, draw) to point the empty object
3. Finally the new keyword returns the object from the constructor function. it is implicitly we no need to write the return code

**Difference between Constructor and Factory function**:

1. In factory function we simply call a function and this function return a new object.
2. But in constructor function we call a function using new operator and this return an object.
3. In factory function we use camel notation but in constructor function we use Pascal notation.

**4) Dynamic Nature of Objects**:

IN JavaScript the objects are dynamic. It means after creating an object we can add or remove property from the object.

**Example**:

const circle = {

radius: 1

};

*//add property*

circle.color = "Red";

circle.draw = function() {};

console.log(circle);

*//{radius: 1, color: "Red", draw: ƒ}*

*//remove property*

delete circle.color;

console.log(circle);

*//{radius: 1, draw: ƒ}*

**Note**:

We cannot re-assignment const variable but we can modify the object.

**5) Constructor Property**:

Every object in JavaScript has a property called constructor and that references the function that was used to construct or create that object. When we create an object using the object literal syntax ({} => object literal), internally the JavaScript engine uses this constructor function.

If we use object literal JavaScript translate it as follow:

let x = {};

let x = new Object();

In JavaScript we have a few other built in constructor. For example, we have

new String() *//'', "", ``*

new Boolean() *// true, false*

new Number() *//1, 2, 3, .......*

**6) Functions are Objects**:

One of the confusing concepts in JavaScript is functions. In JavaScript functions are objects.

function Circle(radius){

*this*.radius = radius;

*this*.draw = function(){

console.log("draw");

}

}

console.log(Circle.name); *//Circle*

console.log(Circle.length); *//1 => numbers of arguments*

Every object in JavaScript has a property called constructor property, and that references the function that was used to create that object.

Circle.constructor

>>function Function()

Here we have another built-in constructor called Function, and when we declare a function using the above syntax, internally, JavaScript engine will use this Function constructor to create this object.

When we declare a function internally it like bellow.

const Circle1 = new Function('radius', `

this.radius = radius;

this.draw = function(){

console.log("draw");;

}

`);

Now we can call Circle1, just like calling Circle function.

const Circle1 = new Function('radius', `

this.radius = radius;

this.draw = function(){

console.log("draw");

}

`);

const circle1 = new Circle1(10);

console.log(circle1); *//Object { radius: 10, draw: draw() }*

**Methods those are available in function**:

function Circle(radius){

*this*.radius = radius;

*this*.draw = function(){

console.log("draw");;

}

}

*//this teference {} the empty object*

*//20 is function argument*

Circle.call({}, 20);

*//the above expression is same as this expression*

*//new operator internally creates {} an empty object and passed as first argument of call method*

const circle = new Circle(10);

If we don’t use the new operator “this” will by defaults point to the global object which is window object.

**7) Value vs. References Type in JavaScript**:

In JavaScript we have two categories of types

1. Value Types
   1. Number
   2. String
   3. Boolean
   4. Symbol (ES6)
   5. undefined
   6. null
2. Reference Type
   1. Object
   2. Function
   3. Array

Primitives and objects behave differently in JavaScript. Primitive are copied by their value but object are copied by their reference.

**Example-1**:

*//Primitives types*

let x = 10;

let y = x;

x = 20;

console.log(x); *//20*

console.log(y); *//10*

*//Reference type*

let m = { value: 10};

let n = m;

m.value = 20;

console.log(m.value); *//20*

console.log(n.value); *//20*

**Example-2**:

*//value*

let number = 20;

function increase(number) {

number++;

console.log(number); *//21*

}

increase(number);

console.log(number); *//20*

**Example-3**:

*//reference*

let obj = {value: 20};

function increase(obj) {

obj.value++;

console.log(obj.value); *//21*

}

increase(obj);

console.log(obj.value); *//21*

**8) Enumerating Properties of an Object**:

For iterate over the property in JavaScript we can follow the following technique.

**for in loop**:

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

for(let key in circle){

console.log(key);

}

*/\*\**

*radius*

*draw*

*\*/*

If we want to get the value of a property we have to use the bracket notation.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

for(let key in circle){

console.log(key, circle[key]);

}

*/\*\**

*radius 1*

*draw function draw()*

*\*/*

**for-of loop**:

We cannot use “for-of” loop directly. If we use we will get error. Because we can use “for-of” loop only on array and maps.

But we can use Object.key() method. This method returns an array of all keys exist in that object and then we can use “for-of” loop.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

for(let key of Object.keys(circle)){

console.log(key);

}

*/\*\**

*radius*

*draw*

*\*/*

**Object.entries method**:

We can also use “Object.entries()” method. This method returns each key-value pair as an array.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

for(let entries of Object.entries(circle)){

console.log(entries);

}

*/\*\**

*Array [ "radius", 1 ]*

*Array [ "draw", draw() ]*

*\*/*

**in operator**:

Sometime we have to see if a given object has a given property or method. To do this we have to use “in” operator.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

if ("radius" in circle){

console.log("yes"); *//yes*

}

**9) Cloning an Object**:

**Iterating an object and pass the property in an empty object**:

By iterating an object we can get all the property of an object and then we can pass the property in an empty object. In this way we can clone an object.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

const another = {};

for(let key in circle){

another[key] = circle[key]; *// another["radius"] = circle["radius"]*

}

console.log(another); *//Object { radius: 1, draw: draw() }*

**Using Object.assign() method**:

Coping and cloning an object is very old approach. In modern JavaScript we have better way to achive the same thing. One of them is “Object.assign()” method.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

const another = Object.assign({}, circle);

console.log(another); *//Object { radius: 1, draw: draw() }*

The targets object no need to require an empty object. It can be an existing object, it can one or more properties or method.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

const another = Object.assign({

color: "red",

}, circle);

console.log(another); *//Object { color: "red", radius: 1, draw: draw() }*

**Using spread operator (…)**:

Using spread operator to cloning an object is a “simple and elegant” way.

const circle = {

radius: 1,

draw() {

console.log('draw');

}

}

const another = {...circle}

console.log(another); *//Object { radius: 1, draw: draw() }*

**10) Garbage Collection**:

Memory management in JavaScript is performed automatically and invisibly to us. When we create object in JavaScript at the time of initialized this object, the memory is automatically allocate to this object, next we can use that, and when we are done using we don’t need to deallocate the memory.

Our JavaScript object has Garbage Collector. The Job of the Garbage Collector is to find the variables or constants that are no longer used and then deallocate the memory that was allocated to them earlier.

5. Object