**02 Objects**

**1) Introduction**:

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*

JavaScript is all about object. In order to learn object-oriented programming, we have to good understanding in the following area.

1. Creating object
2. Factories and Constructor
3. Primitives and Reference Type
4. Working with properties
5. Private properties
6. Getter and Setter method

**2) Object Literals**:

In JavaScript the object literal syntax is {}. Object literal is a simple way to define an object. An object in JavaScript is a key-value pairs. In object we have properties and method. The property is use to hold values and method is use to define some logics.

We can access the members of an object by using dot notation (.).

**Example**:

let circle = {

radious: 1,

location: {

x: 1,

y: 2

},

isVisible: true,

draw: function() {

console.log("draw");

}

};

circle.draw(); *//draw*

**3) Factories**:

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() }*

**4) Constructors**:

Constructor functions are the equivalent of classes in many programming languages. Sometimes people will refer to them 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. By default, “this” point to the global object. If we are running this code inside a browser the global object is the window object and if we are running the code inside Node environment the global object is global
3. Finally, the new keyword returns the object from the constructor function. it is implicitly we no need to write the return code

**Example**:

“this” point to the object when use “new” keyword. But without “new” it points to global window object

function Circle(radius) {

console.log("this", *this*);

*this*.radius = radius;

*this*.draw = function() {

console.log("draw");

};

}

const circle = new Circle(30); *//this Circle {}*

const circle1 = Circle(30); *//Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}*

**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.

**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 Reference Types**:

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 is 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) Adding or Removing Properties**:

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

**Example**:

Add property in circle using dot (.) notation.

function Circle(radius) {

*this*.radius = radius;

*this*.draw = function() {

console.log("draw");

};

}

const circle = new Circle(20);

circle.location = { x: 1 };

console.log(circle); *//Circle {radius: 20, draw: ƒ, location: {…}}*

**Example**:

Add property in circle using bracket ([]) notation.

function Circle(radius) {

*this*.radius = radius;

*this*.draw = function() {

console.log("draw");

};

}

const circle = new Circle(20);

*//add property*

circle["location"] = { x: 1 };

console.log(circle); *//Circle {radius: 20, draw: ƒ, location: {…}}*

Bracket notation is useful in some case. For example, when we want to dynamically access property name. And when we use special character or space in our property name.

**Example**:

Dynamically access property name

function Circle(radius) {

*this*.radius = radius;

*this*.draw = function() {

console.log("draw");

};

}

const circle = new Circle(20);

*//add property*

const propertyName = "location";

circle[propertyName] = { x: 1 };

console.log(circle);

*/\**

*Circle {radius: 20, draw: ƒ, location: {…}}*

*draw: ƒ ()*

*location: {x: 1}*

*radius: 20*

*\_\_proto\_\_: Object*

*\*/*

**Example**:

Use special character in property name.

function Circle(radius) {

*this*.radius = radius;

*this*.draw = function() {

console.log("draw");

};

}

const circle = new Circle(20);

*//add property*

const propertyName = "center-location";

*//circle.center-location => not working*

circle[propertyName] = { x: 1 };

console.log(circle);

*/\**

*Circle {radius: 20, draw: ƒ, location: {…}}*

*draw: ƒ ()*

*location: {x: 1}*

*radius: 20*

*\_\_proto\_\_: Object*

*\*/*

**9) Enumerating Properties**:

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*

}

**10) Abstraction**:

Here we have a circle object and here we add some complexity.

function Circle(radius) {

*this*.radius = radius;

*this*.defaultLocation = {x: 0, y: 0};

*this*.computeOptimumLocation = function(){

*// ...*

}

*this*.draw = function() {

*this*.computeOptimumLocation();

console.log("draw");

};

}

const circle = new Circle(20);

Here in circle we have four members.

1. radius
2. defaultLocation
3. computeOptimumLocation
4. draw

By creating a circle object we can access the entire member. But here is a problem; all the member of the circle object is accessible as a consumer or client of this object. But all the member of the object should not be accessible for consumer or client of this object. It may increase complexity.

For example the "computeOptimumLocation" method should be accessible only inside "draw" method. If "computeOptimumLocation" is accessible outside "draw" method maybe this will put this object in a bad state. After that if we call draw method we will get error in Runtime.

By using abstraction we can overcome this complexity. We can make accessible only radius and draw and hide computeOptimumLocation and defaultLocation.

**11) Private Properties and Methods**:

Inside a constructor function “this” keyword refers to the new created object. If we declare a variable inside an object, the property is not a part of that object, because we don’t set the property in the object. By this technique we can hide any property of an object.

**Example**:

*//only radius and draw is access through circle object*

function Circle(radius) {

*this*.radius = radius;

let defaultLocation = {x: 0, y: 0};

let computeOptimumLocation = function(){

*// ...*

}

*this*.draw = function() {

computeOptimumLocation();

defaultLocation();

console.log("draw");

};

}

const circle = new Circle(20);

Here “computeOptimumLocation” and “defaultLocation” is accessible in draw function for closures in JavaScript.

**Closures concept in JavaScript**:

Here “Circle” is a function and “draw” is another function inside circle. If we declare any variable inside “draw” function then the scope of the variable is limited only inside “draw” function. After execution of “draw” function x and y will go out of scope.

In contrast (বিপরীতে) to scope we have closure. A closure determines what variables will be accessible to an inner function. So “draw” function is able to access all the local variables defined inside it, as well as the variables defines in its parent function (Circle).

Hence “computeOptimumLocation” and “defaultLocation” is accessible inside “draw” function.

Scope is temporary but closure stays there. After executing “draw” function “computeOptimumLocation” and “defaultLocation” stay in memory. They will preserve there state, because they are part of the closure of this draw function.

**12) Getters and Setters**:

A getter or setter can be very useful at some times, but they only need to be used when their specific functionality is required - otherwise plain property access without a getter or setter can be just fine.

A getter has use when we want to run some code every time a property is requested. In our example, the getter always returns an uppercase version of the name regardless of what the case of the name is, while reserving the actual case for internal use.

A setter has use when we want to run some code every time a property is set. In our case, it prevents the setting of a falsey name. You can't implement either of those features without a getter/setter.

Direct property access is a perfectly fine way to do things when we don't need getter or setter special logic.

It's also possible that getters and setters may get and set some property that is not publicly available (so not available at all via a normal property access), either because it's stored somewhere differently (not as a property on this object) or stored privately or even that it's stored else such as in some hardware device. In these cases, the getter and setter simulate the value being in a property when it actually isn't. So, they simplify the interface while allowing the actual value to be stored or retrieved from anywhere.

**Example**:

class Employee {

constructor(name, age) {

*this*.\_name = name;

*this*.age = age;

}

doWork() {

return `${*this*.name} is working`;

}

get name() {

return *this*.\_name.toUpperCase();

}

set name(newName) {

if (newName) {

*this*.\_name = newName;

}

}

}

let man = new Employee("Ruhul", 30);

console.log(man.name, man.age); *//RUHUL 30*

man.name = "Shamim";

man.age = 28;

console.log(man.name, man.age); *//SHAMIM 28*

**13) Constructors**:

02 Objects