**Part 1 – JavaScript Basics**

JavaScript is a lightweight, cross-platform, object-oriented programming language. Today, JavaScript can be used in different places

* Client-side: Used in browser
* Server-side: Node.js, On the server

**Primitive Data Type**

* **Number** – Floating point numbers, for decimals and integers
* **String** – Sequence of characters, used for text
* **Boolean** – Logical data type that can be only **true** or **false**
* **Undefined** – Data type of a variable that does not have a value yet
* **Null** – Means **non-existent**

**Variable mutation and type coercion**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Variable mutation and type coercion

\*/

let firstName = 'John';

let age = 28;

// Type coercion

console.log(firstName + ' ' + age);

let job, isMarried;

job = 'teacher';

isMarried = false;

console.log(firstName + ' is a ' + age + ' year old ' + job + '. Is he married? ' + isMarried);

// Variable mutation

age = 'twenty eight';

job = 'driver';

alert(firstName + ' is a ' + age + ' year old ' + job + '. Is he married? ' + isMarried);

let lastName = prompt('What is his last Name?');

console.log(firstName + ' ' + lastName);

**Function Statements and Expression**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Function Statements and Expressions

\*/

//Function Statements

function funcName(year, firstName) {}

funcName(1990, 'John');

// Function expression

let funcName = function(job, firstName) {}

funcName('teacher', 'John')

**JavaScript Array**

//Array Basics

const arr = ['A', 'B', 'C']

console.log(arr.length)//3

console.log(arr[0])//A

console.log(arr[arr.length-1])//C

arr.push('D')//push method - will add element in the last - 'D', ['A', 'B', 'C', 'D']

arr.pop() //pop method - will remove last element - 'D', ['A', 'B', 'C']

arr.shift()//shift method - will remove first element -'A', ['B', 'C']

arr.unshift('A')//unshift method - will add element at first place. 'A', ['A', 'B', 'C']

//splice(start, deleteCount, item1, item2, ...itemN)

arr.splice(0, 1, 'Aa','Bb','Cc')//A, Will delete element at 0 index and will and new Item 'Aa' at 0 index.

//Original Array - ['Aa', 'Bb', 'Cc', 'B', 'C']

arr.splice(1, 2)// ['Bb', 'Cc']. Original Array - ['Aa', 'B', 'C']

arr.splice(0,1,'A') //['Aa'] Original Array - ['A', 'B', 'C']

arr.indexOf('D')//-1 Original Array - ['A', 'B', 'C']

arr.indexOf('C')//2 Original Array - ['A', 'B', 'C']

arr.join('-')//join method - will convert array to sting 'A-B-C'. It does not affect Original Array - ['A', 'B', 'C']

arr.slice(1)// ['B', 'C']

arr.slice(2, 3)// ['C']

arr.slice(-2)// ['B', 'C']

const arr2 = ['X', 'Y', 'Z']

arr.concat(arr2)//Concate 2 array

**Objects and Properties**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Objects and properties

\*/

// Object literal

var john = {

    firstName: 'John',

    lastName: 'Smith',

    birthYear: 1992,

    family: ['Jane', 'Mark', 'Bob', 'Emily'],

    job: 'teacher',

    isMarried: false,

    calcAge: function() {

        this.age = 2018 - this.birthYear;

    }

};

console.log(john.firstName);

console.log(john['lastName']);

var x = 'birthYear';

console.log(john[x]);

john.job = 'designer';

john['isMarried'] = true;

console.log(john);

// new Object syntax

var jane = new Object();

jane.firstName = 'Jane';

jane.birthYear = 1969;

jane['lastName'] = 'Smith';

console.log(jane);

john.calcAge();

console.log(john);

**Loops and iteration**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Loops and iteration

\*/

// for loop

var john = ['John', 'Smith', 1990, 'designer', false, 'blue'];

for (var i = 0; i < john.length; i++) {

    console.log(john[i]);

}

// While loop

var i = 0;

while(i < john.length) {

    console.log(john[i]);

    i++;

}

// continue and break statements

var john = ['John', 'Smith', 1990, 'designer', false, 'blue'];

for (var i = 0; i < john.length; i++) {

    if (typeof john[i] !== 'string') continue;

    console.log(john[i]);

}

for (var i = 0; i < john.length; i++) {

    if (typeof john[i] !== 'string') break;

    console.log(john[i]);

}

// Looping backwards

for (var i = john.length - 1; i >= 0; i--) {

    console.log(john[i]);

}

Final Example –

var john = {

    fullName: 'John Smith',

    bills: [124, 48, 268, 180, 42],

    calcTips: function() {

        this.tips = [];

        this.finalValues = [];

        for (var i = 0; i < this.bills.length; i++) {

            // Determine percentage based on tipping rules

            var percentage;

            var bill = this.bills[i];

            if (bill < 50) {

                percentage = .2;

            } else if (bill >= 50 && bill < 200) {

                percentage = .15;

            } else {

                percentage = .1;

            }

            // Add results to the corresponing arrays

            this.tips[i] = bill \* percentage;

            this.finalValues[i] = bill + bill \* percentage;

        }

    }

}

var mark = {

    fullName: 'Mark Miller',

    bills: [77, 475, 110, 45],

    calcTips: function() {

        this.tips = [];

        this.finalValues = [];

        for (var i = 0; i < this.bills.length; i++) {

            // Determine percentage based on tipping rules

            var percentage;

            var bill = this.bills[i];

            if (bill < 100) {

                percentage = .2;

            } else if (bill >= 100 && bill < 300) {

                percentage = .1;

            } else {

                percentage = .25;

            }

            // Add results to the corresponing arrays

            this.tips[i] = bill \* percentage;

            this.finalValues[i] = bill + bill \* percentage;

        }

    }

}

function calcAverage(tips) {

    var sum = 0;

    for (var i = 0; i < tips.length; i++) {

        sum = sum + tips[i];

    }

    return sum / tips.length;

}

// Do the calculations

john.calcTips();

mark.calcTips();

john.average = calcAverage(john.tips);

mark.average = calcAverage(mark.tips);

console.log(john, mark);

if (john.average > mark.average) {

    console.log(john.fullName + '\'s family pays higher tips, with an average of $' + john.average);

} else if (mark.average > john.average) {

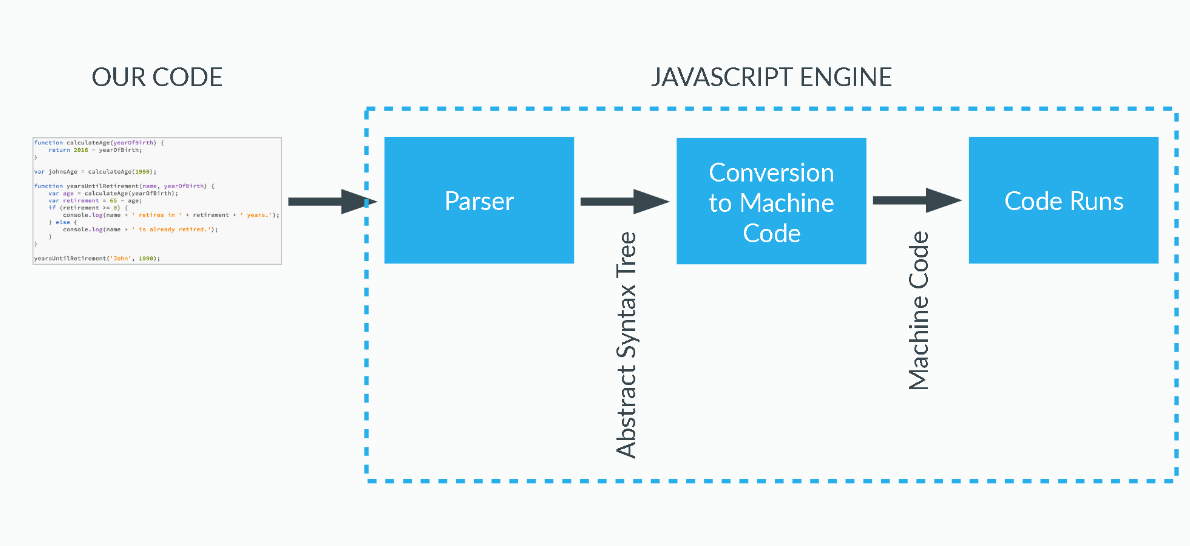
    console.log(mark.fullName + '\'s family pays higher tips, with an average of $' + mark.average);

}

**How JavaScript code is Executed ?**

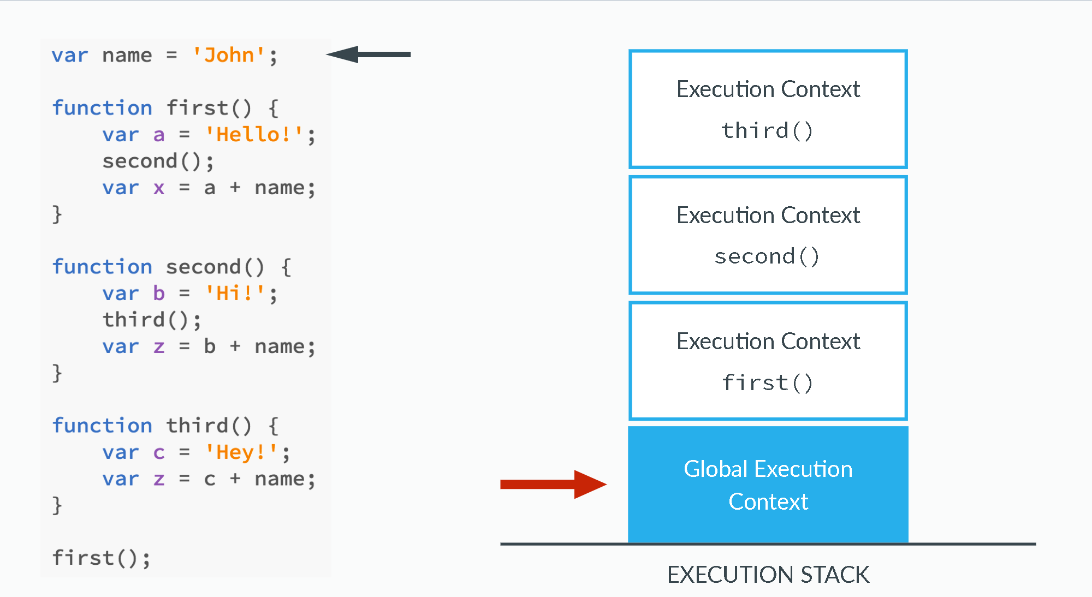
JavaScript is always hosted in some environment. And that is most typically a browser such as Google Chrome, Firefox, Safari, etc. And on server node JS.  
JavaScript engine (Such as Google's V8 engine, that is used in Google Chrome) that takes our code and executes it. A JavaScript engine is a program that executes JavaScript code.

the first thing that happens inside the engine, is that our code is parsed by a parser, which basically reads our code line by line, and checks if the syntax of the code that we gave it, is correct. So this means that the parser knows the JavaScript rules and how it has to be written in order to be correct to be valid.  
then the parser produces a data structure known as the Abstract Syntax Tree, which is then translated into machine code.  
So this code is no longer JavaScript code, but a code, or let's say a set of instructions, that can be executed directly by the computer's processor.



**Execution Context**

A box, A container or A wrapper which stores variables/methods and in which a piece of code is evaluated and executed.  
The default is **Global Execution Context**. Code that is not inside any function associated with Global Object. In the browser Global object is window object.

lastName === window.lastName //true****

**Execution Context Creation Phases**

This is creation phases –

1. Creation of the Variable Object
2. Creation of the Scope Chain
3. Determine value of ‘this’ variable

Creation of the Variable Object –

* The argument object is created, containing all the arguments that were passed into function
* Code is scanned for function declarations: for each function, a property is created in the variable Object, pointing to the function.
* Code is scanned for variable declarations: for each variable, a property is created in the variable object and set to undefined.

**Hoisting**

// Lecture: Hoisting

// functions

calculateAge(1965);

function calculateAge(year) {

    console.log(2016 - year);

}

// retirement(1956);//undefined

var retirement = function(year) {

    console.log(65 - (2016 - year));

}

// variables

console.log(age);//undefined

var age = 23;

function foo() {

    console.log(age);//undefined

    var age = 65;

    console.log(age);

}

foo();

console.log(age);

Creation of the Scope Chain/ Lexical Scope of a variable

// Lecture: Scoping

// First scoping example

var a = 'Hello!';

first();

function first() {

    var b = 'Hi!';

    second();

    function second() {

        var c = 'Hey!';

        console.log(a + b + c);

    }

}

// Example to show the differece between execution stack and scope chain

var a = 'Hello!';

first();

function first() {

    var b = 'Hi!';

    second();

    function second() {

        var c = 'Hey!';

        third()

    }

}

function third() {

    var d = 'John';

    //console.log(c);

    console.log(a+d);

}

Determine value of ‘this’ variable

// Lecture: The this keyword

//console.log(this);

calculateAge(1985);

function calculateAge(year) {

    console.log(2016 - year);

    console.log(this);

}

var john = {

    name: 'John',

    yearOfBirth: 1990,

    calculateAge: function() {

        console.log(this);

        console.log(2016 - this.yearOfBirth);

        function innerFunction() {

            console.log(this);

        }

        innerFunction();

    }

}

john.calculateAge();

var mike = {

    name: 'Mike',

    yearOfBirth: 1984

};

mike.calculateAge = john.calculateAge;

mike.calculateAge();

**Execution Context Execution Phases**

The code of the function that generated the current Execution Context is ran line by line.

**Part 2 – JavaScript Advanced**

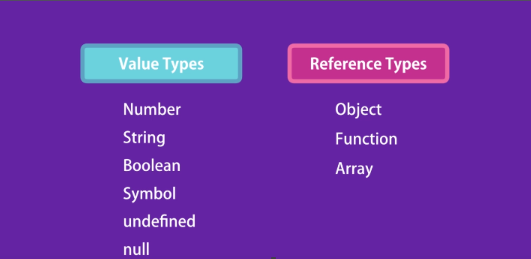
**Non Primitives/Reference Types**

There are essentially two types of values in JavaScript. The first type is primitives, and the second type is objects (which also includes functions). Primitive values include simple value types such as numbers (which includes everything from integers to floats to Infinity to NaN), booleans, strings, undefined, and null (note: even though typeof null === 'object', null is a still primitive value).

Primitive values are also immutable. They can’t be changed. Of course, a variable with a primitive assigned can be reassigned. For example, when you write the code let x = 1; x++;, you've reassigned the variable x. But, you haven't mutated the primitive numeric value of 1.

Some languages, such as C, have the concept of pass-by-reference and pass-by-value. JavaScript sort of has this concept too, though, it’s inferred based on the type of data being passed around. If you ever pass a value into a function, reassigning that value will not modify the value in the calling location. However, if you modify a non-primitive value, the modified value will also be modified where it has been called from.

Primitives are copied by their value And Object are copied by their reference

****

//Value Type Example

let x = 10

let y = x

x = 20

console.log(x) // 20

console.log(y) // 10

//Reference Type Example

let x = {

    value: 10

}

let y = x

x.value = 20

console.log(x) // {value:20}

console.log(y) // {value:20}

let number = 10

function increase(number) {

    number++

}

increase(number)

console.log(number); //10

let obj = { value: 10 };

function increase(obj) {

    obj.value++

}

increase(obj)

console.log(obj); //{value: 11}

const obj1 = { name: "Intrinsic" };

const obj2 = { name: "Intrinsic" };

console.log(obj1 === obj2); // false

// Though, their .name properties ARE primitives:

console.log(obj1.name === obj2.name); // true

Primitives are copied by their value And Object are copied by their

**Object Constructor Function**

Constructor functions technically are regular functions. There are two conventions though:

* They are named with capital letter first.
* They should be executed only with "new" operator.

function Person(name, yearOfBirth, job) {

    this.name = name;

    this.yearOfBirth = yearOfBirth;

    this.job = job;

}

OR

let Person = function(name, yearOfBirth, job) {

    this.name = name;

    this.yearOfBirth = yearOfBirth;

    this.job = job;

}

var john = new Person('John', 1990, 'teacher');

var jane = new Person('Jane', 1969, 'designer');

When a function is executed with new, it does the following steps:

1. A new empty object is created and assigned to this.
2. The function body executes. Usually it modifies this, adds new properties to it.
3. The value of this is returned.

In a example it looks like –

function Person(name, yearOfBirth, job) {

    //1. this = {};  (implicitly)

    //2. add properties to this

    this.name = name;

    this.yearOfBirth = yearOfBirth;

    this.job = job;

    //3. return this;  (implicitly)

}

If we have many lines of code all about creation of a single complex object, we can wrap them in an immediately called constructor function, like this:

// create a function and immediately call it with new

let person = new function() {

    this.name = "John";

    this.yearOfBirth = 1990;

    this.job = "teacher";

    // ...other code for user creation

    // maybe complex logic and statements

    // local variables etc

};

Inside a function, we can check whether it was called with new or without it, using a special new.target property.

function Person(name, yearOfBirth, job) {

    if (!new.target) { // if you run me without new

        return new Person(name, yearOfBirth, job); // ...I will add new for you

      }

    this.name = name;

    this.yearOfBirth = yearOfBirth;

    this.job = job;

}

var john = Person('John', 1990, 'teacher');

Usually, constructors do not have a return statement. Their task is to write all necessary stuff into this, and it automatically becomes the result.

But if there is a return statement, then the rule is simple:

* If return is called with an object, then the object is returned instead of this.
* If return is called with a primitive, it’s ignored.

In other words, return with an object returns that object, in all other cases this is returned.

For instance, here return overrides this by returning an object:

function BigUser() {

    this.name = "John";

    return { name: "Godzilla" };  // <-- returns this object

  }

  new BigUser().name;  // Godzilla, got that object

function SmallUser() {

    this.name = "John";

    return; // <-- returns this

}

new SmallUser().name;  // John

By the way, we can omit parentheses after new, if it has no arguments:

let user = new User; // <-- no parentheses

// same as

let user = new User();

Methods in constructor

function User(name) {

    this.name = name;

    this.sayHi = function() {

        alert( "My name is: " + this.name );

    };

}

let john = new User("John");

john.sayHi(); // My name is: John

**Object Constructor vs Object Literals**

The regular {...} syntax (Object Literals Syntax) allows us to create one object. But Using Constructor function we can create many similar object

**Prototypes**

The prototype is an object that is associated with every Array, functions and objects by default in JavaScript. Every object (except the root object) has a prototype (parent).  
To get the prototype of an object:

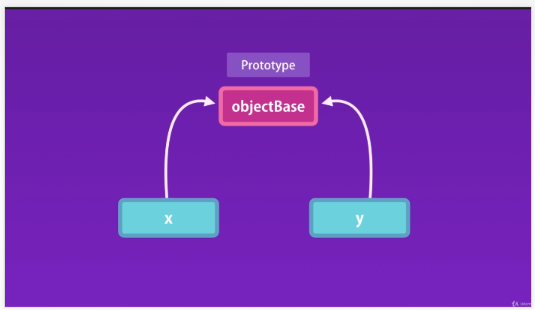
Object.getPrototypeOf(obj);

In console, you can inspect "\_\_proto\_\_" property. But you should not use that in the code.

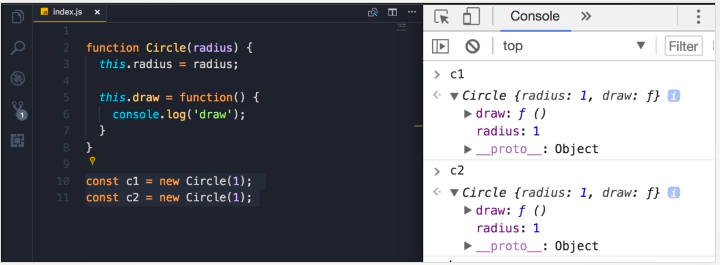
const x = {}, const y = {}

Object.getPrototypeOf(x) === Object.getPrototypeOf(y) //true

x.\_\_proto\_\_ === y.\_\_proto\_\_ //true //depricated use only in console

****

Problem with creating objects with the constructor function:



On executing the above code, the JavaScript engine will create two copies of the constructor function, each for c1 and c2.

Every object created using the constructor function will have its own copy of properties and methods. It doesn’t make sense to have two instances of function ***draw*** that do the same thing. Storing separate instances of function for each object results in wastage of memory.

When a function is created in JavaScript, the JavaScript engine adds a prototype property to the function. This prototype property is an object (called a prototype object) that has a constructor property by default. The constructor property points back to the function on which prototype object is a property. We can access the function’s prototype property using functionName.prototype.

function Circle(radius) {

    //Instance members

    this.radius = radius

    //Instance methods

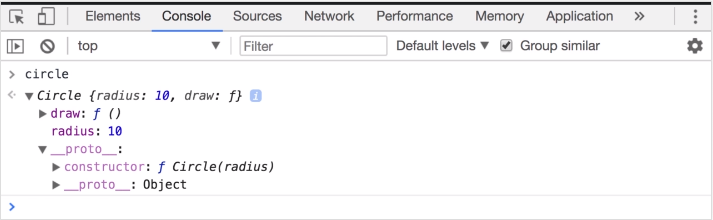
    this.draw = function() {

        console.log('draw')

    }

}

const circle = new Circle(1)



As shown in the above image, Circle constructor function has a prototype property that points to the prototype object. The prototype object has a constructor property that points back to the Circle constructor function.

As seen from the above image prototype property of the function is an object (prototype object) with two properties:

* constructor property which points to Circle function itself
* \_\_proto\_\_ property: We will discuss this while explaining inheritance in JavaScript

When an object is created in JavaScript, JavaScript engine adds a \_\_proto\_\_ property to the newly created object which is called dunder proto. dunder proto or \_\_proto\_\_ points to the prototype object of the constructor function.

Example of Human constructor

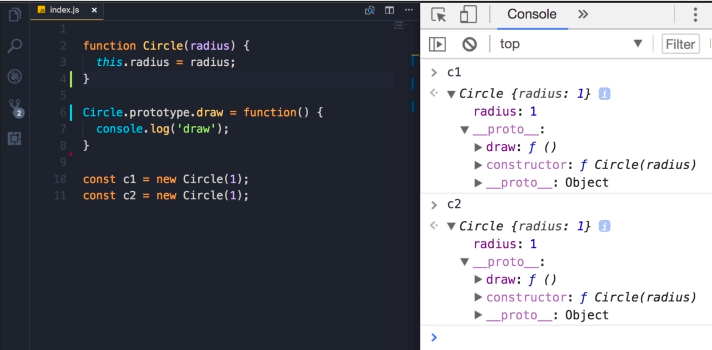


Human.prototype === person1.\_\_proto\_\_ //true

Object created by a given constructor will have the same prototype

person1.\_\_proto\_\_ === person2.\_\_proto\_\_ //true

c1.\_\_proto\_\_ === c2.\_\_proto\_\_ //true



function Circle(radius) {

    //Instance members

    //Instance methods

    this.radius = radius

    this.move = function() {

        this.draw();

        consol e.log('move')

    }

}

//Prototype members

Circle.prototype.draw = function() {

    //this.move();

    console.log('draw')

}

const c1 = new Circle(1)

const c2 = new Circle(2)

c1.toString() // "[object Object]"

Circle.prototype.toString = function() {

    return 'Circle with radius ' + this.radius;

}

c1.toString() // "Circle with radius 1"

function Circle(radius) {

    //Instance members

    this.radius = radius

    //Instance methods

    this.move = function() {

        console.log('move')

    }

}

const c1 = new Circle(1)

//Prototype members

Circle.prototype.draw = function() {

    console.log('draw')

}

c1.draw() //will still work

//Returns instance members

Object.keys(c1) // (2) ["radius", "move"]

//Returns all members (instance and prototype)

for (let key in c1) console.log(key) // radius, move, draw

c1.hasOwnProperty('radius') //true

c1.hasOwnProperty('draw') //false – because this is prototype property

//Create an empty constructor function

function Person(){}

//Add property name, age to the prototype property of the Person constructor function

Person.prototype.name = "Ashwin" ;

Person.prototype.age = 26;

Person.prototype.sayName = function(){

    console.log(this.name);

}

//Create an object using the Person constructor function

var person1 = new Person();

//Access the name property using the person object

console.log(person1.name)// Output" Ashwin

As we can see that person1 object is empty and it does not have any property except its dunder proto property. So how does the output of console.log(person.name) was Ashwin?

When we try to access a property of an object, the JavaScript engine first tries to find the property on the object, if the property is present on the object it outputs its value. But, if the property is not present on the object then it tries to find the property on the prototype object or dunder proto of the object. If the property is found the value is returned else JavaScript engine tries to find the property on the dunder proto of the object. This chain continues until the dunder proto property is null. In these cases, the output will be undefined.

So, when person1.name is called, JavaScript engine checks if the property exists on the person1 object. In this case, name property was not on the person1’s object. So, now JavaScript engine checks if the name property exists on the dunder proto property or the prototype of the person1’s object. In this case, name property was there on the dunder proto property or the prototype of person1’s object. Hence, the output was returned Ashwin.

Problems with the prototype

As prototype object is shared among all the objects created using the constructor function, its properties and methods are also shared among all the objects. If an object A modifies a property of the prototype having primitive value, other objects will not get affected by, as object A will create a property on its objects as shown below.

console.log(person1.name);//Output: Ashwin

console.log(person2.name);//Output: Ashwin

person1.name = "Ganguly"

console.log(perosn1.name);//Output: Ganguly

console.log(person2.name);//Output: Ashwin

Here (line 1 and 2), both person1 and person2 does not have name property, hence they access the prototypes name property and hence the output is same for both.

When person1 want to have a different value for the name property, it creates a name property on its object.

//Create an empty constructor function

function Person(){

}

//Add property name, age to the prototype property of the Person constructor function

Person.prototype.name = "Ashwin" ;

Person.prototype.age = 26;

Person.prototype.friends = ['Jadeja', 'Vijay'],//Arrays are of reference type in JavaScript

Person.prototype.sayName = function(){

    console.log(this.name);

}

//Create objects using the Person constructor function

var person1= new Person();

var person2 = new Person();

//Add a new element to the friends array

person1.friends.push("Amit");

console.log(person1.friends);// Output: "Jadeja, Vijay, Amit"

console.log(person2.friends);// Output: "Jadeja, Vijay, Amit"

In the above example, person1 and person2 point to the same friends' array of the prototype object. person1 modifies friends property by adding another string in the array.

As friends' array exists on Person.prototype, not on person1, the changes made in the friend’s property by person1 object is reflected on person2.friends also (which points to the same array).

If the intention is to have an array shared by all instances, then this outcome is okay. But here this was not the case.

***To solve the problems with the prototype and the problems with the constructor, we can combine both the constructor and function***

//Define the object specific properties inside the constructor

function Human(name, age){

    this.name = name,

    this.age = age,

    this.friends = ["Jadeja", "Vijay"]

}

//Define the shared properties and methods using the prototype

Human.prototype.sayName = function(){

    console.log(this.name);

}

//Create two objects using the Human constructor function

var person1 = new Human("Virat", 31);

var person2 = new Human("Sachin", 40);

//Lets check if person1 and person2 have points to the same instance of the sayName function

console.log(person1.sayName === person2.sayName) // true

//Let's modify friends property and check

person1.friends.push("Amit");

console.log(person1.friends)// Output: "Jadeja, Vijay, Amit"

console.log(person2.friends)//Output: "Jadeja, Vijay"

You can modify built in java script method using prototype –

//Don't modify built in javascript method by using prototype

Array.prototype.shuffle = function() {

    // ..

}

const arr = []

arr.shuffle()

**Prototypical Inheritance**

Let’s define a shape object and put duplicate method there, and then have circle and square inherited from the shape object.

Let's define two objects here, one is a shape and the other is a circle

function Shape() {}

Shape.prototype.duplicate = function() {

    console.log("Shape Duplicate")

}

function Circle(radius) {

    this.radius = radius

}

//Circle.prototype = Object.create(Object.prototype); // In memory with every Object Base is associated

Circle.prototype = Object.create(Shape.prototype)

Circle.prototype.draw = function() {

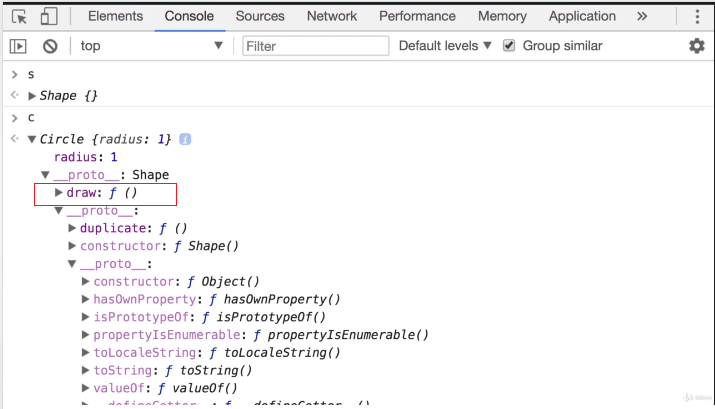
    console.log('Circle Draw')

}

const s = new Shape()

const c = new Circle(1)

Now check and console c variable in console –



Now check c.duplicate() // Shape Duplicate

Now there is a tiny problem with this implementation. Go back to the console. Earlier, I told you that every object in JavaScript has a constructor property, that returns the function that was used to construct

or create that object. Let's take a look at our circle object. So, the circle, look at it's prototype, this is essentially circle.prototype, now here we have the constructor property which

references our circle function. Now with this, technically we can create a circle object like this. Circle.prototype, that returns this object here, right, and here we have this constructor property,

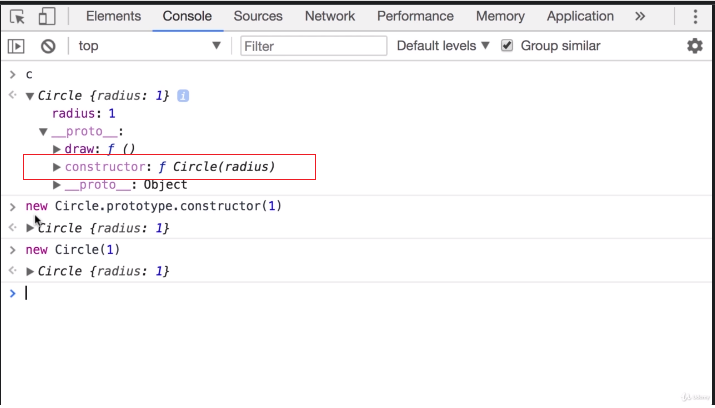
and this returns this circle constructor function, right? So technically we can new up this expression and pass a radius of 1 and this returns a circle object. So this expression is equivalent to new Circle

of 1. They're exactly the same. Now obviously we use this form because it's shorter and cleaner, but in some rare circumstances you may get a constructor function somewhere in your application, and you may want to dynamically create an object based on that constructor function. So there you can access the prototype property and from there you get the constructor and then use the new operator.

Circle.prototype.constructor = Circle

new Circle.prototype.constructor()

new Circle()



When you are resetting prototype You need to reset constructor as well So Add this line -

Circle.prototype.constructor = Circle

Calling Super Constructor

function Shape(color) {

    this.color = color

}

Shape.prototype.duplicate = function() {

    console.log("Shape Duplicate")

}

function Circle(radius, color) {

    //Shape(color) //will not work. Circle will not have color property

    this.radius = radius

}

//Circle.prototype = Object.create(Object.prototype); // In memory with every Object Base is associated

Circle.prototype = Object.create(Shape.prototype)

Circle.prototype.constructor = Circle

Circle.prototype.draw = function() {

    console.log('Circle Draw')

}

const s = new Shape('blue')

const c = new Circle(1, 'red')

Remember when you use the new operator, three things happen, this new operator –

* creates a new empty object
* sets this to point to that object. So here we set the latest property to that new object
* new object will be returned from this constructor

If you don't use the new operator this by default will point to the global object which is window in the browser and global in node. So the reason this didn't work.

Here we're calling the shape function, and by default, this, in this function, will point to a global object, so we didn't set the color property on the new instance of the circle object, we set it on the window object, let's verify that. So, window.color, look, it's red.

***Every function in JavaScript is an object.***

function Shape(color) {

    this.color = color

}

Shape.prototype.duplicate = function() {

    console.log("Shape Duplicate")

}

function Circle(radius, color) {

    //Shape(color) //will not work. Circle will not have color property

    Shape.call(this, color)

    this.radius = radius

}

//Circle.prototype = Object.create(Object.prototype); // In memory with every Object Base is associated

Circle.prototype = Object.create(Shape.prototype)

Circle.prototype.constructor = Circle

Circle.prototype.draw = function() {

    console.log('Circle Draw')

}

const s = new Shape('blue')

const c = new Circle(1, 'red')

Intermediate Function Inheritance

function Shape() {}

Shape.prototype.duplicate = function() {

    console.log("Shape Duplicate")

}

function Circle(radius) {

    this.radius = radius

}

Circle.prototype = Object.create(Shape.prototype)

Circle.prototype.constructor = Circle

Circle.prototype.draw = function() {

    console.log('Circle Draw')

}

function Square(size) {

    this.size = size

}

Square.prototype = Object.create(Shape.prototype)

Square.prototype.constructor = Square

const s = new Shape()

const c = new Circle(1)

const sq = new Square(10)

Method Overriding

function Shape() {}

Shape.prototype.duplicate = function() {

    console.log("Shape Duplicate")

}

function Circle(radius) {

    this.radius = radius

}

Circle.prototype = Object.create(Shape.prototype)

Circle.prototype.constructor = Circle

Circle.prototype.duplicate = function() {

    Shape.prototype.duplicate.call(this)

    console.log('Circle duplicate')

}

function Square(size) {

    this.size = size

}

Square.prototype = Object.create(Shape.prototype)

Square.prototype.constructor = Square

Square.prototype.duplicate = function() {

    Shape.prototype.duplicate.call(this)

    console.log('Square duplicate')

}

const s = new Shape()

const c = new Circle(1)

const sq = new Square(10)

**Closures**

A closure can be defined as a JavaScript feature in which the inner function has access to the outer function variable. In JavaScript, every time a closure is created with the creation of a function.

The closure has three scope chains listed as follows:

* Access to its own scope.
* Access to the variables of the outer function.
* Access to the global variables.

**Bind, Call and Apply**

Traditionally in JavaScript, you can have objects that have their own properties and methods. For example, object1 cannot use the methods of object2 and vice versa.

var car = {

    registrationNumber: "GA12345",

    brand: "Toyota",

    displayDetails: function(){

        console.log(this.registrationNumber + " " + this.brand);

    }

}

car.displayDetails(); // GA12345 Toyota

But what if we want to borrow a method?

var myCarDetails =  car.displayDetails;

myCarDetails();

Well, this won’t work as the “this” will be now assigned to the global context which doesn’t have neither the registrationNumber nor the brand property.

But there’s a way to overcome this restriction.

var myCarDetails = car.displayDetails.bind(car);

myCarDetails(); // GA12345 Toyota

You can use call(), apply(), and bind() methods to tie a function into an object and call the function as if it belonged to that object.

**Syntax: objectMethod.call(otherObject, objectMethod function params**)

Function.call(object, function params)

var obj = { num: 2 };

function add(a, b){

  return this.num + a + b;

}

console.log(add.call(obj, 3, 5));

function Circle(radius) {

    //Instance members

    this.radius = radius

    //Instance methods

    this.draw = function() {

        console.log('draw')

    }

}

const circle = new Circle(1)

//OR

Circle.call({}, 1) // This is Same as const circle = new Circle(1)

Here first param {} is empty object refer to 'this'

Second param 1 refer to number of argument we pass in Circle constructor method

If param is n in number then we will pass like this –

//Circle.call({}, 1, 2, 3, ..upto n)

If without ***new*** keyword how it will work –

const circle2 =  Circle(1)

Circle.call(window, 1)

Now first param will point to window/Global object

**Apply** same as *call* but param will pass as a array

//Circle.call({}, [1, 2, 3, ..upto n])

**Bind** same as *call* but it return a function when you want to print value then execute that function with passing parameters - 1, 2, 3, ..upto n

**Part 3 – JavaScript ES6**

**var, let and const**

**var** The scope of the var keyword is the global or function scope. It means variables defined outside the function can be accessed globally, and variables defined inside a particular function can be accessed within the function.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-1\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var a = 10

function f() {

    console.log(a) //10

}

f();

console.log(a); //10

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-2\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function f() {

    // It can be accessible any where within this function

    var a = 10;

    console.log(a) //10

}

f();

// A cannot be accessible outside of function

console.log(a); //ReferenceError: a is not defined

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-3\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

console.log(a); //undefined

var a = 10;

Problem with var

var greeter = "hey hi";

var times = 4;

if (times > 3) {

    var greeter = "say Hello instead";

}

console.log(greeter) // "say Hello instead"

//AND

var a = 10

// User can re-declare variable using var in the middle of file

var a = 8

// User can update var variable

a = 7

**let** is an improved version of the var keyword. The scope of a let variable is only block scoped. It can’t be accessible outside the particular block ({block}).

/\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let a = 10;

function f() {

    let b = 9

    console.log(b); //9

    console.log(a); //10

}

f();

/\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let a = 10;

function f() {

    if (true) {

        let b = 9

        console.log(b); //9

    }

    // It gives error as it defined in if block

    console.log(b); //ReferenceError: b is not defined

}

f()

console.log(a) //10

/\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-3\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var a = 10

// User can re-declare variable using var

var a = 8 //Uncaught SyntaxError: Identifier 'a' has already been declared

// User can update variable

a = 7

/\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-4\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let a = 10

if (true) {

    let a = 9

    console.log(a) // 9

}

console.log(a) // 10

/\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-5\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

console.log(a); //Uncaught ReferenceError: Cannot access 'a' before initialization

let a = 10;

**const** The const keyword has all the properties that are the same as the let keyword, It is block scoped except the user cannot update it.

When users declare a const variable, they need to initialize it, otherwise, it returns an error. The user cannot update the const variable once it is declared

/\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const a = 10;

function f() {

    a = 9

    console.log(a) //TypeError:Assignment to constant variable.

}

f();

/\*\*\*\*\*\*\*\*\*\*\*\*\*Ex-2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const a = {

    prop1: 10,

    prop2: 9

}

// It is allowed

a.prop1 = 3

// It is not allowed

a = {

    b: 10,

    prop2: 9

}

//Uncaught SyntaxError:Unexpected identifier

**Default Parameters**

let calculateArea = function(height = 50, width = 80) {

    // write logic

    ...

}

**Create Your Own Map function**

**The Rest Operator**

The rest parameter syntax allows us to represent an indefinite number of arguments as an array. With the help of a rest parameter a function can be called with any number of arguments.

function fun(...input){

    //Note: input argument will be treat like an array [1,2,3,4,5]

    let sum = 0;

    for(let i of input){

        sum+=i;

    }

    return sum;

}

console.log(fun(1,2)); //3

console.log(fun(1,2,3)); //6

console.log(fun(1,2,3,4,5)); //15

**The Spread Operator**

* Object spread operator (...) unpacks the own enumerable properties of an object.
* Object spread operator can be used to clone an object or merge objects into one. The cloning is always shallow.
* When merging objects, the spread operator defines new properties while the Object.assign() assigns them.

let numbers = [1,2,3,4]

console.log(...numbers)

//1

//2

//3

//4

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example-1\*\*\*\*\*\*\*\*\*\*\*\*/

let rgb = ['red', 'green', 'blue'];

let cmyk = ['cyan', 'magenta', 'yellow', 'black'];

let merge = [...rgb, ...cmyk];

console.log(merge);

//Output: [ 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'black' ]

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example-2\*\*\*\*\*\*\*\*\*\*\*\*/

const circle = {

    radius: 10

};

const coloredCircle = {

    ...circle,

    color: 'black'

};

console.log(coloredCircle);

//Output: {

//     radius: 10,

//     color: 'black'

// }

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example-3\*\*\*\*\*\*\*\*\*\*\*\*/

const circle = {

    radius: 10

};

const clonedCircle = {

    ...circle

};

console.log(clonedCircle);

//Output: { radius: 10 }

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example-4\*\*\*\*\*\*\*\*\*\*\*\*/

const circle = {

    radius: 10,

    style: {

        color: 'blue'

    }

};

const clonedCircle = {

    ...circle

};

clonedCircle.style = 'red';

console.log(clonedCircle);

//Output: { radius: 10, style: 'red' }

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example-5\*\*\*\*\*\*\*\*\*\*\*\*/

const circle = {

    radius: 10

};

const style = {

    backgroundColor: 'red'

};

const solidCircle = {

    ...circle,

    ...style

};

console.log(solidCircle);

//Output: { radius: 10, backgroundColor: 'red' }

**Spread operator vs. Object.assign()** The spread operator (...) defines new properties in the target object while the Object.assign() method assigns them. It has two side effects.

The Object.assign() invokes setters on the target object while the spread operator doesn’t. The following illustrates how to clone an object using both Object.assign() and spread operator (...). However, only the Object.assign() method triggers the setters:

class Circle {

    constructor(radius) {

        this.radius = radius;

    }

    set diameter(value) {

        this.radius = value / 2;

        console.log('SET ', value);

    }

    get diameter() {

        return this.radius \* 2;

    }

}

let circle = new Circle(100);

let cloneCircle1 = Object.assign(circle, {

    diameter: 200

});

let cloneCircle2 = {

    ...circle

};

If a target object has a read-only property, you cannot use Object.assign() method to assign a new value to that property. However, the spread operator ( ...) can define a new property. Suppose you have an object called blueSquare whose the color property is readonly:

const blueSquare = {

    length: 100,

    color: 'blue'

};

Object.defineProperty(blueSquare, 'color', {

    value: 'blue',

    enumerable: true,

    writable: false

});

console.log(blueSquare);

//Output: { length: 100, color: 'blue' }

// merge style and blueSquare objects:

const style = {

    color: 'green'

};

const greenSquare = {

    ...blueSquare,

    ...style

};

console.log(greenSquare);

//Output: { length: 100, color: 'green' }

// merge style and redSquare objects: ERROR

const redSquare = Object.assign(blueSquare, {

    color: 'red'

});

//Output: TypeError: Cannot assign to read only property 'color' of object '#<Object>'

**The for-of loop**

ES6 introduced a new statement for...of that iterates over an iterable object such as:

* Built-in Array, String, Map, Set, …
* Array-like objects such as arguments or NodeList
* User-defined objects that implement the iterator protocol.

for (variable of iterable) {

    // ...

 }

/\*\*\*\*\*\*\*\*\*\*Example - 1\*\*\*\*\*\*\*\*\*\*\*\*/

let scores = [80, 90, 70];

for (const score of scores) {

    console.log(score);

}

//Output: 80 90 70

/\*\*\*\*\*\*\*\*\*\*Example - 2\*\*\*\*\*\*\*\*\*\*\*\*/

let colors = ['Red', 'Green', 'Blue'];

for (const [index, color] of colors.entries()) {

    console.log(`${color} is at index ${index}`);

}

//Output:

// Red is at index 0

// Green is at index 1

// Blue is at index 2

/\*\*\*\*\*\*\*\*\*\*Example - 3\*\*\*\*\*\*\*\*\*\*\*\*/

const ratings = [{

        user: 'John',

        score: 3

    },

    {

        user: 'Jane',

        score: 4

    },

    {

        user: 'David',

        score: 5

    },

    {

        user: 'Peter',

        score: 2

    },

];

let sum = 0;

for (const {

        score

    }

    of ratings) {

    sum += score;

}

console.log(`Total scores: ${sum}`); // 14

/\*\*\*\*\*\*\*\*\*\*Example - 4\*\*\*\*\*\*\*\*\*\*\*\*/

let str = 'abc';

for (let c of str) {

    console.log(c);

}

//Output: a b c

/\*\*\*\*\*\*\*\*\*\*Example - 5\*\*\*\*\*\*\*\*\*\*\*\*/

let colors = new Map();

colors.set('red', '#ff0000');

colors.set('green', '#00ff00');

colors.set('blue', '#0000ff');

for (let color of colors) {

    console.log(color);

}

//Output:

// [ 'red', '#ff0000' ]

// [ 'green', '#00ff00' ]

// [ 'blue', '#0000ff' ]

/\*\*\*\*\*\*\*\*\*\*Example - 6\*\*\*\*\*\*\*\*\*\*\*\*/

let nums = new Set([1, 2, 3]);

for (let num of nums) {

    console.log(num);

}

/\*\*\*\*\*\*\*\*\*\*Example - 7\*\*\*\*\*\*\*\*\*\*\*\*/

(function() {

    for (const argument of arguments) {

        console.log(argument);

    }

})(1, 2, 3);

// 1

// 2

// 3

/\*\*\*\*\*\*\*\*\*\*Example - 8\*\*\*\*\*\*\*\*\*\*\*\*/

const articleParagraphs = document.querySelectorAll('article > p');

for (const paragraph of articleParagraphs) {

    paragraph.classList.add('read');

}

/\*\*\*\*\*\*\*\*\*\*Example - 9\*\*\*\*\*\*\*\*\*\*\*\*/

function\* foo() {

    yield 1;

    yield 2;

    yield 3;

};

for (const o of foo()) {

    console.log(o);

    break; // closes iterator, execution continues outside of the loop

}

console.log('done');

**(Fat) Arrow Functions**

The fat arrows are amazing because they would make your this behave properly, i.e., this will have the same value as in the context of the function— it won’t mutate.

$('.btn').click((event) => {

    this.doSomething()

 });

**Destructuring**

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const [a, b] = array;

const [a, , b] = array;

const [a = aDefault, b] = array;

const [a, b, ...rest] = array;

const [a, , b, ...rest] = array;

const [a, b, ...{ pop, push }] = array;

const [a, b, ...[c, d]] = array;

const { a, b } = obj;

const { a: a1, b: b1 } = obj;

const { a: a1 = aDefault, b = bDefault } = obj;

const { a, b, ...rest } = obj;

const { a: a1, b: b1, ...rest } = obj;

let a, b, a1, b1, c, d, rest, pop, push;

[a, b] = array;

[a, , b] = array;

[a = aDefault, b] = array;

[a, b, ...rest] = array;

[a, , b, ...rest] = array;

[a, b, ...{ pop, push }] = array;

[a, b, ...[c, d]] = array;

({ a, b } = obj); // brackets are required

({ a: a1, b: b1 } = obj);

({ a: a1 = aDefault, b = bDefault } = obj);

({ a, b, ...rest } = obj);

({ a: a1, b: b1, ...rest } = obj);

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let person = {

    firstName: 'John',

    lastName: 'Doe'

};

let { firstName: fname, lastName: lname } = person;

let { firstName, lastName } = person;

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 3\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const x = [1, 2, 3, 4, 5];

const [y, z] = x;

console.log(y); // 1

console.log(z); // 2

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 4\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const [firstElement, secondElement] = list;

// is equivalent to:

// const firstElement = list[0];

// const secondElement = list[1];

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 5\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const foo = ['one', 'two', 'three'];

const [red, yellow, green] = foo;

console.log(red); // "one"

console.log(yellow); // "two"

console.log(green); // "three"

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 6\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let a, b;

[a, b] = [1, 2];

console.log(a); // 1

console.log(b); // 2

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 7\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const foo = ['one', 'two'];

const [red, yellow, green, blue] = foo;

console.log(red); // "one"

console.log(yellow); // "two"

console.log(green); // undefined

console.log(blue);  //undefined

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 8\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let a, b;

[a=5, b=7] = [1];

console.log(a); // 1

console.log(b); // 7

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 9\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let a = 1;

let b = 3;

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

console.log(a); // 3

console.log(b); // 1

const arr = [1,2,3];

[arr[2], arr[1]] = [arr[1], arr[2]];

console.log(arr); // [1,3,2]

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 10\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function f() {

    return [1, 2];

}

let a, b;

[a, b] = f();

console.log(a); // 1

console.log(b); // 2

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 11\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function f() {

    return [1, 2, 3];

}

const [a, , b] = f();

console.log(a); // 1

console.log(b); // 3

const [c] = f();

console.log(c); // 1

[,,] = f();

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 12\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const [a, ...b] = [1, 2, 3];

console.log(a); // 1

console.log(b); // [2, 3]

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 13\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const [a, ...b,] = [1, 2, 3];

// SyntaxError: rest element may not have a trailing comma

// Always consider using rest operator as the last element

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 14\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const [a, b, ...{ pop, push }] = [1, 2];

console.log(a, b); // 1 2

console.log(pop, push); // [Function pop] [Function push]

const [a, b, ...[c, d]] = [1, 2, 3, 4];

console.log(a, b, c, d); // 1 2 3 4

/\*\*\*\*\*\*\*\*\*\*\*\*\*Example - 15\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const [a, b, ...[c, d, ...[e, f]]] = [1, 2, 3, 4, 5, 6];

console.log(a, b, c, d, e, f); // 1 2 3 4 5 6

const { a, ...{ b } } = { a: 1, b: 2 };

// SyntaxError: `...` must be followed by an identifier in declaration contexts

let a, b;

({ a, ...{ b } } = { a: 1, b: 2 });

// SyntaxError: `...` must be followed by an assignable reference in assignment contexts

**Template Literals**

let name = `Your name is ${firstName} ${lastName}.`

Multi-line Strings

let poemData = `Johny Johny Yes Papa,

                Eating sugar?  No, papa!

                Telling lies? No, papa!

                Open your mouth Ah, ah, ah!`

Enhanced Object Literals If key and value are same then we can write one one value.

function getLaptop(make, model, year) {

    return {

       make,

       model,

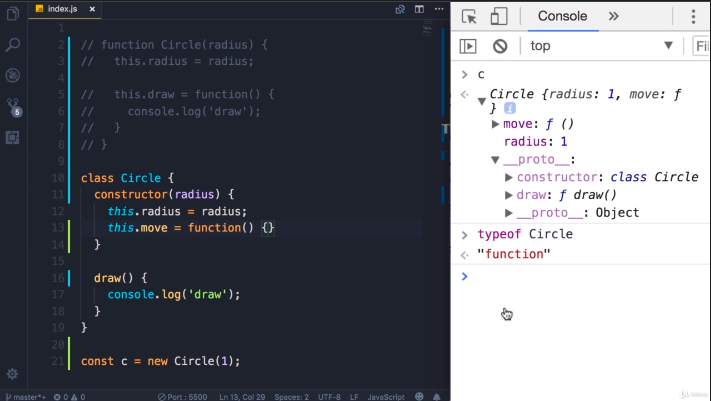
       year

    }

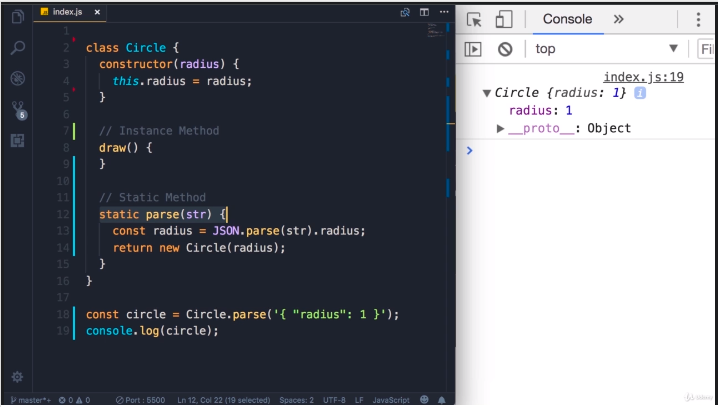
 }

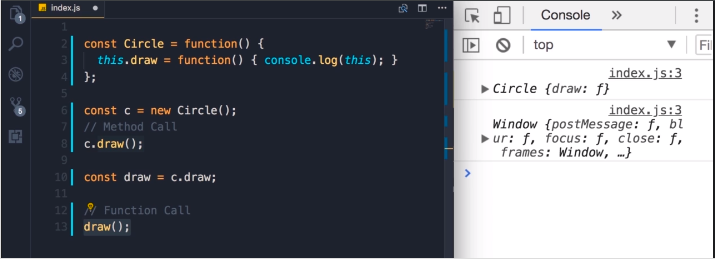
 getLaptop("Apple", "MacBook", "2015");

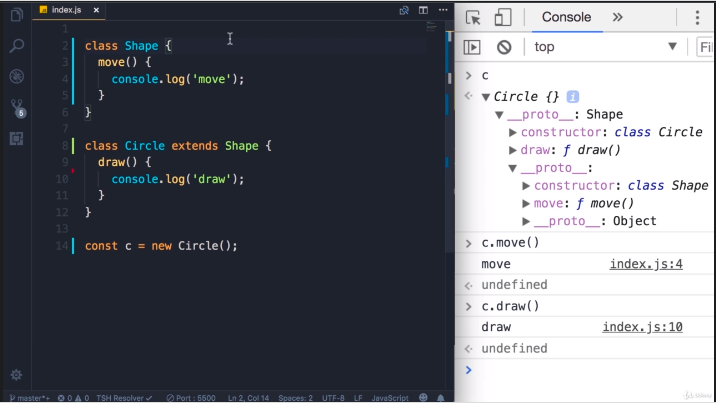
**Classes**

****

Static Methods

****

****

****

**Modules**

**Symbols**

The JavaScript ES6 introduced a new primitive data type called Symbol. Symbols are immutable (cannot be changed) and are unique.

* A Symbol() method always return a unique value.
* A symbol value may be used as an identifier for object properties.
* Symbols are immutable, just like numbers or strings.
* Symbols cannot be typecasted to primitive data types.

// two symbols with the same description

const value1 = Symbol('hello');

const value2 = Symbol('hello');

console.log(value1 === value2); // false

You use the Symbol() function to create a Symbol

// creating symbol

const x = Symbol()

typeof x; // symbol

const x = Symbol('hey');

console.log(x); // Symbol(hey)

//Access Symbol Description

const x = Symbol('hey');

console.log(x.description); // hey

Add Symbol as an Object Key

let id = Symbol("id");

let person = {

    name: "Jack",

    // adding symbol as a key

    [id]: 123 // not "id": 123

};

console.log(person); // {name: "Jack", Symbol(id): 123}

Symbols are not included in for...in Loop

let id = Symbol("id");

let person = {

    name: "Jack",

    age: 25,

    [id]: 12

};

// using for...in

for (let key in person) {

    console.log(key);

}

// name age

If the same code snippet is used in various programs, then it is better to use Symbols in the object key. It's because you can use the same key name in different codes and avoid duplication issues. For example,

let person = {

    name: "Jack"

};

// creating Symbol

let id = Symbol("id");

// adding symbol as a key

person[id] = 12;

In the above program, if the person object is also used by another program, then you wouldn't want to add a property that can be accessed or changed by another program. Hence by using Symbol, you create a unique property that you can use.

Now, if the other program also needs to use a property named id, just add a Symbol named id and there won't be duplication issues. For example,

let person = {

    name: "Jack"

};

let id = Symbol("id");

person[id] = "Another value";

In the above program, even if the same name is used to store values, the Symbol data type will have a unique value.

In the above program, if the string key was used, then the later program would have changed the value of the property. For example,

let person = {

    name: "Jack"

};

// using string as key

person.id = 12;

console.log(person.id); // 12

// Another program overwrites value

person.id = 'Another value';

console.log(person.id); // Another value

In the above program, the second user.id overwrites the previous value.

**Iterators and Generators**

It’s an object or pattern that allows us to traverse over a list or collection. Iterators define the sequences and implement the iterator protocol that returns an object by using a next() method that contains the value and done. The value contains the next value of iterator sequence and the done is the boolean value true or false if the last value of the sequence has been consumed then it’s true else false. We can check if any entity is by default iterable or not

We can check its prototype and can see if it is having a method **Symbol(Symbol.iterator)** or not. In **Array.prototype** you will find **Symbol(Symbol.iterator): ƒ values()** method. Array is by default iterable. Also, String, Map & Set are built-in iterables because their prototype objects all have a **Symbol.iterator()** method.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const array = ['a', 'b', 'c'];

const it = array[Symbol.iterator]();

// and on this iterator method we have ‘next’ method

document.write(JSON.stringify(it.next()));

//{ value: "a", done: false }

document.write(JSON.stringify(it.next()));

//{ value: "b", done: false }

document.write(JSON.stringify(it.next()));

//{ value: "c", done: false }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const array = ['a', 'b', 'c'];

const it = array[Symbol.iterator]();

for (let value of it) {document.write(value)}//abc

**Iterable protocol:** The object must define a method with ‘Symbol.iterator’ the key which returns an object which itself follows iterator protocol. The object must define ‘next’ method which returns an object having two properties ‘value’ and ‘done’

var iterable = {

    i: 0,

    [Symbol.iterator]() {

      var that = this;

      return {

        next() {

          if (that.i < 5) {

            return { value: that.i++, done: false }

          } else {

            return { value: undefined, done: true }

          }

        }

      }

    }

  }

  for(let value of iterable){document.write(value)}

  //Output: 0 1 2 3 4

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Sequence {

    constructor(start = 0, end = Infinity, interval = 1) {

            this.start = start;

            this.end = end;

            this.interval = interval;

        }

        [Symbol.iterator]() {

            let counter = 0;

            let nextIndex = this.start;

            return {

                next: () => {

                    if (nextIndex <= this.end) {

                        let result = {

                            value: nextIndex,

                            done: false

                        }

                        nextIndex += this.interval;

                        counter++;

                        return result;

                    }

                    return {

                        value: counter,

                        done: true

                    };

                }

            }

        }

};

let evenNumbers = new Sequence(2, 10, 2);

for (const num of evenNumbers) {

    console.log(num);

}

// Output:

// 2

// 4

// 6

// 8

// 10

let evenNumbers = new Sequence(2, 10, 2);

let iterator = evenNumbers[Symbol.iterator]();

let result = iterator.next();

while (!result.done) {

    console.log(result.value);

    result = iterator.next();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Sequence {

    constructor(start = 0, end = Infinity, interval = 1) {

            this.start = start;

            this.end = end;

            this.interval = interval;

        }

        [Symbol.iterator]() {

            let counter = 0;

            let nextIndex = this.start;

            return {

                next: () => {

                    if (nextIndex <= this.end) {

                        let result = {

                            value: nextIndex,

                            done: false

                        }

                        nextIndex += this.interval;

                        counter++;

                        return result;

                    }

                    return {

                        value: counter,

                        done: true

                    };

                },

                return: () => {

                    console.log('cleaning up...');

                    return {

                        value: undefined,

                        done: true

                    };

                }

            }

        }

}

let oddNumbers = new Sequence(1, 10, 2);

for (const num of oddNumbers) {

    if (num > 7) {

        break;

    }

    console.log(num);

}

//Output

// 1

// 3

// 5

// 7

// cleaning up...

**Generator Advantages of generators:**

They are memory efficient as lazy evaluation takes place, i.e, delays the evaluation of an expression until its value is needed.

Use-case (generators)

* writing generators in redux-saga
* async-await (Implemented with promise and generators)

**Generator-Function** : A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. The yield statement suspends function’s execution and sends a value back to caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run.

// An example of generator function

function\* gen(){

    yield 1;

    yield 2;

    ...

    ...

}

**Generator-Object** : Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for of” loop (as shown in the above program)

The Generator object is returned by a generating function and it conforms to both the iterable protocol and the iterator protocol.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function\* fun() {

    yield 10;

    yield 20;

    yield 30;

}

// Calling the Generate Function

var gen = fun();

console.log(gen.next().value);

console.log(gen.next().value);

console.log(gen.next().value);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function\* nextNatural() {

    var naturalNumber = 1;

    // Infinite Generation

    while (true) {

        yield naturalNumber++;

    }

}

// Calling the Generate Function

var gen = nextNatural();

// Loop to print the first

// 10 Generated number

for (var i = 0; i < 10; i++) {

    // Generating Next Number

    console.log(gen.next().value);

}

//Output: 1 2 3 4 5 6 7 8 9 10

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var array = ['a', 'b', 'c'];

function\* generator(arr) {

    let i = 0;

    while (i < arr.length) {

        yield arr[i++]

    }

}

const it = generator(array);

// we can do it.return() to finish the generator

**Encountering yield and yield\* syntax**

**yield**: pauses the generator execution and returns the value of the expression which is being written after the yield keyword.

**yield\***: it iterates over the operand and returns each value until done is true.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const arr = ['a', 'b', 'c'];

function\* generator() {

    yield 1;

    yield\* arr;

    yield 2;

}

for (let value of generator()) {

    console.log(value);

}

//Output : 1 a b c 2

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var createOwnIterable = {

    \*[Symbol.iterator]() {

        yield 'a';

        yield 'b';

        yield 'c';

    }

}

for (let value of createOwnIterable) {

    console.log(value);

}

//Output: a b c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function\* generator() {

    yield 'a';

    return 'result';

    yield 'b';

}

var it = generator();

console.log(JSON.stringify(it.next()));

// {value: "a", done: false}

console.log(JSON.stringify(it.next()));

// {value: "result", done: true}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 4 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function\* firstGenerator() {

    yield 2;

    yield 3;

}

function\* secondGenerator() {

    yield 1;

    yield\* firstGenerator();

    yield 4;

}

for (let value of secondGenerator()) {

    console.log(value)

}

//Output: 1 2 3 4

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 5 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function\* generator() {

    ['a', 'b', 'c'].forEach(value => yield value) // This will give syntax error

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 6 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const firstPromise = () => {

    return new Promise((resolve, reject) => {

        setTimeout(() => resolve(1), 5000)

    })

}

const secondPromise = () => {

    return new Promise((resolve, reject) => {

        setTimeout(() => resolve(2), 3000)

    })

}

async function\* generator() {

    const firstPromiseResult = await firstPromise();

    yield firstPromiseResult;

    const secondPromiseResult = await secondPromise();

    yield secondPromiseResult;

}

var it = generator();

for await (let value of it) {

    console.log(value);

}

//Output: (after 5 seconds) 1  (after 3 seconds) 2

**Maps & Sets**

**The Reflect API**

**The Proxy API**

**Part 4 – Important Points Could Not Covered**

**IIFE**

An Immediately-invoked Function Expression (IIFE for friends) is a way to execute functions immediately, as soon as they are created. IIFEs are very useful because they don't pollute the global object, and they are a simple way to isolate variables declarations.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

(function() {

    /\* ... \*/

})();

//Arrow function IIFE

(() => {

    /\* ... \*/

})();

//async IIFE

(async () => {

    /\* ... \*/

})();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var budgetController = (function() {

    var x = 23

    var add = function(a) {

        return x + a;

    }

    return {

        publicTest: function(b) {

            console.log(x)

            console.log(add(b))

        }

    }

})()

var UIController = (function() {

    //Some Code

})();

var controller = (function(budgetCtrl, UICtrl) {

    var z = budgetCtrl.publicTest(8)

    return {

        anotherPublic: function() {

            console.log(z)

        }

    }

})(budgetController, UIController)

**Curry Function**

It is a technique in functional programming, transformation of the function of multiple arguments into several functions of a single argument in sequence.

***A function from callable as f(a, b, c) into callable as f(a)(b)(c).***

We simply wrap function inside a function, which means we are going to return a function from another function to obtain this kind of translation. The parent function takes the first provided argument and returns the function that takes the next argument and this keeps on repeating till the number of arguments ends. Hopefully, the function that receives the last argument returns the expected result.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function getProduct(num1) {

    return function(num2) {

        return num1 \* num2;

    };

}

getProduct(10)(20);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function getTravelTime(distance) {

    return function(speed) {

        return distance / speed;

    };

}

const travelTimeBosNyc = getTravelTime(400);

const travelTimeMiamiAtlanta = getTravelTime(600);

console.log(travelTimeBosNyc(100));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function curry(f) { // curry(f) does the currying transform

    return function(a) {

        return function(b) {

            return f(a, b);

        };

    };

}

// usage

function sum(a, b) {

    return a + b;

}

let curriedSum = curry(sum);

alert(curriedSum(1)(2)); // 3

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 4 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function curry(func) {

    return function curried(...args) {

        if (args.length >= func.length) {

            return func.apply(this, args);

        } else {

            return function(...args2) {

                return curried.apply(this, args.concat(args2));

            }

        }

    };

}

function sum(a, b, c) {

    return a + b + c;

}

let curriedSum = curry(sum);

alert(curriedSum(1, 2, 3)); // 6, still callable normally

alert(curriedSum(1)(2, 3)); // 6, currying of 1st arg

alert(curriedSum(1)(2)(3)); // 6, full currying

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 5 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// func is the function to transform

function curried(...args) {

    if (args.length >= func.length) { // (1)

        return func.apply(this, args);

    } else {

        return function(...args2) { // (2)

            return curried.apply(this, args.concat(args2));

        }

    }

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 6 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function curry(f) {

    return function currify() {

        const args = Array.prototype.slice.call(arguments);

        return args.length >= f.length ?

            f.apply(null, args) :

            currify.bind(null, ...args)

    }

}

**Shallow vs Deep Cloning**

**Shallow Copy:** When a reference variable is copied into a new reference variable using the assignment operator, a shallow copy of the referenced object is created. In simple words, a reference variable mainly stores the address of the object it refers to. When a new reference variable is assigned the value of the old reference variable, the address stored in the old reference variable is copied into the new one. This means both the old and new reference variable point to the same object in memory. As a result if the state of the object changes through any of the reference variables it is reflected for both.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const person = {

    firstName: 'John',

    lastName: 'Doe'

};

// using spread ...

let person1 = {

    ...person

}; // Shallow copy

// using  Object.assign() method

let person2 = Object.assign({}, person); // Shallow copy

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var employee = {

    eid: "E102",

    ename: "Jack",

    eaddress: "New York",

    salary: 50000

}

console.log("employee => ", employee);

var employee2 = employee;    // Shallow copy

console.log("employee2 => ", employee2);

console.log("---------After modification----------");

employee2.ename = "Beck";

console.log("employee => ", employee);

console.log("employee2 => ", employee2);

// Name of the employee as well as employee2 is changed.

In the below example When you show the values of the person object, you will find that the address information changed but the first name does not change. The reason is that the address is reference value while the first name is a primitive value. Both person and copiedPerson references different objects but these objects reference the same address objects.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let person = {

    firstName: 'John',

    lastName: 'Doe',

    address: {

        street: 'North 1st street',

        city: 'San Jose',

        state: 'CA',

        country: 'USA'

    }

};

let copiedPerson = Object.assign({}, person);

copiedPerson.firstName = 'Jane'; // disconnected

copiedPerson.address.street = 'Amphitheatre Parkway'; // connected

copiedPerson.address.city = 'Mountain View'; // connected

console.log(copiedPerson);

//Output: {

//     firstName: 'Jane',

//     lastName: 'Doe',

//     address: {

//         street: 'Amphitheatre Parkway',

//         city: 'Mountain View',

//         state: 'CA',

//         country: 'USA'

//     }

// }

console.log(person);

//Output: {

//     firstName: 'John',

//     lastName: 'Doe',

//     address: {

//         street: 'Amphitheatre Parkway',

//         city: 'Mountain View',

//         state: 'CA',

//         country: 'USA'

//     }

// }

**Deep Copy**: Unlike the shallow copy, deep copy makes a copy of all the members of the old object, allocates separate memory location for the new object and then assigns the copied members to the new object. In this way, both the objects are independent of each other and in case of any modification to either one the other is not affected. Also, if one of the objects is deleted the other still remains in the memory. Now to create a deep copy of an object in JavaScript we use JSON.parse() and JSON.stringify() methods.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var employee = {

    eid: "E102",

    ename: "Jack",

    eaddress: "New York",

    salary: 50000

}

console.log("=========Deep Copy========");

var newEmployee = JSON.parse(JSON.stringify(employee));

console.log("Employee=> ", employee);

console.log("New Employee=> ", newEmployee);

console.log("---------After modification---------");

newEmployee.ename = "Beck";

newEmployee.salary = 70000;

console.log("Employee=> ", employee);

console.log("New Employee=> ", newEmployee);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

let person = {

    firstName: 'John',

    lastName: 'Doe',

    address: {

        street: 'North 1st street',

        city: 'San Jose',

        state: 'CA',

        country: 'USA'

    }

};

let copiedPerson = JSON.parse(JSON.stringify(person));

copiedPerson.firstName = 'Jane'; // disconnected

copiedPerson.address.street = 'Amphitheatre Parkway';

copiedPerson.address.city = 'Mountain View';

console.log(person);

//Output: {

//     firstName: 'John',

//     lastName: 'Doe',

//     address: {

//         street: 'North 1st street',

//         city: 'San Jose',

//         state: 'CA',

//         country: 'USA'

//     }

// }

**Array reduce function**

Parameter: This method accepts five parameters as mentioned below:

**function(total, currentValue, index, arr)**: It is the required parameter and used to run for each element of array. It contains four parameter which are listed below:

**total:** It is required parameter and used to specify the initialValue, or the previously returned value of the function.

**currentValue:** It is required parameter and used to specify the value of the current element.

**currentIndex:** It is optional parameter and used to specify the array index of the current element.

**arr:** It is optional parameter and used to specify the array object the current element belongs to.

**initialValue:** It is optional parameter and used to specify the value to be passed to the function as the initial value.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Callback function

reduce(callbackFn)

reduce(callbackFn, initialValue)

// Inline callback function

reduce(function(previousValue, currentValue) { /\* ... \*/ })

reduce(function(previousValue, currentValue, currentIndex) { /\* ... \*/ })

reduce(function(previousValue, currentValue, currentIndex, array) { /\* ... \*/ })

reduce(function(previousValue, currentValue) { /\* ... \*/ }, initialValue)

reduce(function(previousValue, currentValue, currentIndex) { /\* ... \*/ }, initialValue)

reduce(function(previousValue, currentValue, currentIndex, array) { /\* ... \*/ }, initialValue)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const array5 = [15, 16, 17, 18, 19];

function reducer(previous, current, index, array5) {

    const returns = previous + current;

    console.log(`previous: ${previous}, current: ${current}, index: ${index}, returns: ${returns}`);

    return returns;

}

array5.reduce(reducer);

// previous: 15, current: 16, index: 1, returns: 31

// previous: 31, current: 17, index: 2, returns: 48

// previous: 48, current: 18, index: 3, returns: 66

// previous: 66, current: 19, index: 4, returns: 85

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const numArray = [1, 2, [3, 10, [11, 12]],

    [1, 2, [3, 4]], 5, 6

];

function flattenArray(data) {

    // our initial value this time is a blank array

    const initialValue = [];

    // call reduce on our data

    return data.reduce((total, value) => {

        // if the value is an array then recursively call reduce

        // if the value is not an array then just concat our value

        return total.concat(Array.isArray(value) ? flattenArray(value) : value);

    }, initialValue);

}

//Output:[1,2,3,10,11,12,1,2,3,4,5,6]

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example - 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const balls = ['green', 'blue', 'pink', 'yellow', 'blue', 'yellow', 'green'];

var ballsMap = balls.reduce((acc, ball) => {

    if (acc[ball] >= 1) acc[ball]++;

    else acc[ball] = 1;

    return acc;

}, {});

console.log(ballsMap);

// Expected output: {green: 2, blue: 2, pink: 1, yellow: 2}

Make a program Double of array all number element

**Garbage Collection**

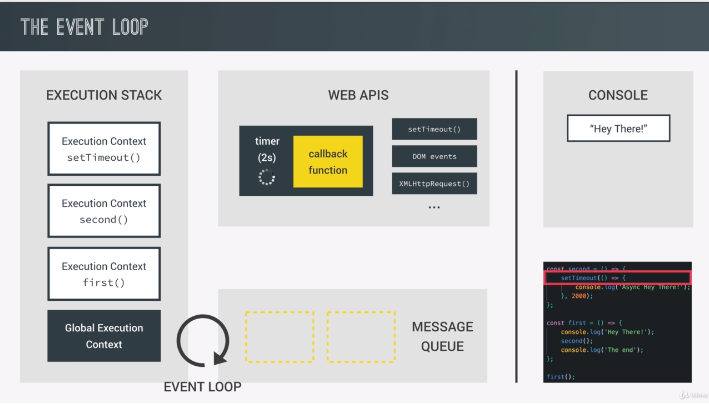
[Garbage Collection](https://javascript.info/garbage-collection)

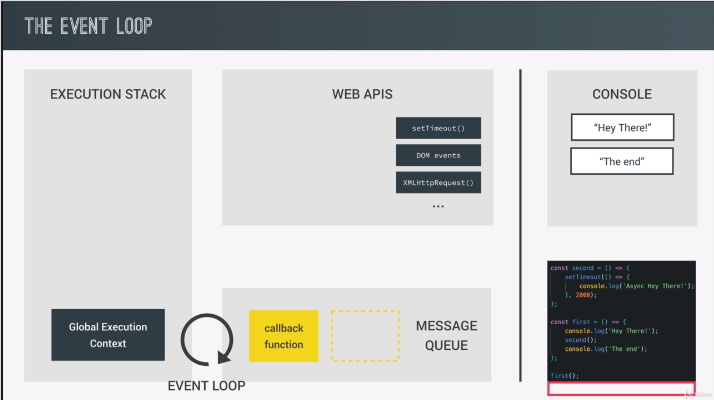
[Garbage Collection](https://medium.com/swlh/garbage-collection-in-javascript-b1e4dbffb51)

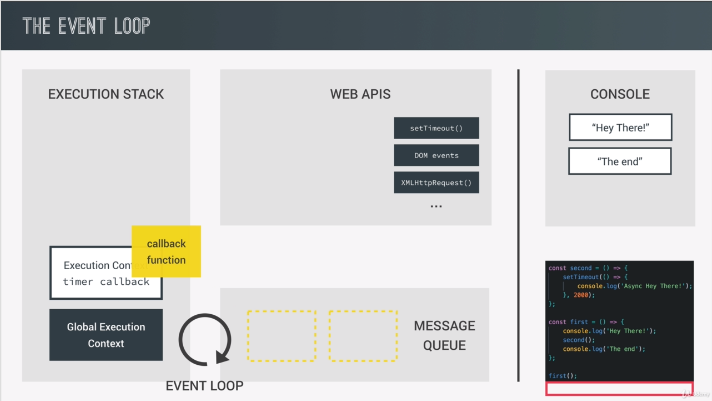
**Event Loop**

**WEB APIs** – are Stuff like DOM manipulation methods, Set Timeout, HTTP requests for AJAX, geolocation, local storage and tons of other things, actually live outside of the JavaScript engine.

**Event Loop** - when set Timeout function is called another Execution Context to be created. Timer will keep running for two seconds, asynchronously of course, so that our code can keep running without being blocked. When we call the Set Timeout function, the timer is created, with our callback function right inside the Web APIs environment. And there it keeps sitting until it finishes its work all in an Asynchronous way. Since the timer keeps working, basically in the background, we don't have to wait, and can keep executing our code. we have executed all our code in a Synchronous way, and have the timer run Asynchronously in the background. Let's suppose that our two seconds have passed and the timer disappears. But what happens to our callback function now? Well, it simply moves to the Message Queue, where it waits to be executed as soon as the Execution Stack is empty. In the case of DOM events our event listeners sit in the Web APIs environment, waiting for a certain event to happen. and as soon as that event then happens, then the callback function is placed on a Message Queue, ready to be executed. The job of the Event Loop is to constantly monitor the Message Queue and the Execution Stack the event loop takes the callback and pushes it onto the stack, where a new Execution Context is created for that function. if there were some callbacks waiting right now, like data coming back from an AJAX request, or the handler of a DOM event, then the Event Loop would continue pushing them onto the stack until all of them were processed.

****

****

****

**Object.defineProperty() Method**

**Static method Object.defineProperty()** defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

Object.defineProperty(obj, prop, descriptor)

**Use Strict**

**Part 6 – Asynchronous JavaScript**

These topics will cover in node js Part

**Callback**

**Callback Hell**

**Promises**

**Async/Await**

**Part 7 – Modern JavaScript**

These topics will cover in React js Part

**The Webpack**

**Babel**