# Basic Javascript

## What is JavaScript?

JavaScript is a dynamic programming language that's used for web development, in web applications, for game development, and lots more. It allows us to implement dynamic features on web pages that cannot be done with only HTML and CSS.

## Data types

Data types basically specify what kind of data can be stored and manipulated within a program.

There are two basic data types in JavaScript which can be divided into two main categories: primitive (or primary), and non-primitive (or reference).

#### Primitive types

Primitive data types can hold only one value at a time. The following are the primitive data types.

**Boolean**: Any of two values: true or false i.e: const isOk = true;

**Null**: Denotes a null value i.e: let a = null;

**Undefined**: A special data type. When a variable is declared but not initialized then its default value is undefined i.e: let a;

**Number**: An integer or a floating-point number i.e: 3, 3.234, 3e-2 etc.

**BigInt**: An integer with arbitrary precision i.e: 900719925124740999n , 1n etc.

**String**: Represents textual data i.e: 'hello world!', "hello world!", and `hello world!`

**Symbol**: Data type whose instances are unique and immutable i.e: let value = Symbol('hello');

#### Non-primitive types

Non-primitive data types can hold collections of values and more complex entities. The following are the non-primitive data types.

**Object**: An object is a collection of the key-value pairs data. i.e: const student = { name: 'Bablu', id: 10 };

**Array**: An array is a type of object used for storing multiple values in a single variable i.e: const colors = ["Red", "Yellow", "Green", "Orange"];

**Function**: The function is a callable object that executes a block of code. Since functions are objects, so it is possible to assign them to variables.

const greeting = function(){

return "Hello World!";

}

console.log(typeof greeting); // function

console.log(greeting()); // Hello World!

greeting.token = 'asdf’;

console.log(greeting.token); // asdf

**Dates**: The Date object is also a built-in JavaScript object. It allows you to get the user's local time by accessing the computer system clock through the browser for example,

const d = new Date();

console.log(d) // Tue Sep 07 2021 01:58:09 GMT+0600 (Bangladesh Standard Time)

**RegExp:** The RegExp object is also a built-in JavaScript object that represents regular expression.

**Class**: Class is also a non-primitive data type.

For more info about the data types here [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)

## Dynamic typing

JavaScript is a dynamically typed (loosely typed) language. JavaScript automatically determines the variables' data type for you.

let foo = 42; // foo is now a number

foo = 'bar'; // foo is now a string

foo = true; // foo is now a boolean

### typeof keyword

To find the type of a variable as a string value, you can use the typeof operator. For example,

const name = 'Bablu';

typeof name; // returns "string"

const number = 10;

typeof number; //returns "number"

const valueChecked = true;

typeof valueChecked; //returns "boolean"

const a = null;

typeof a; // returns "object"

**Note**: Notice that typeof returned "object" for the null type. This is a known issue in JavaScript since its first release.

## Concurrency Model and Event Loop

**Event Loop:** The event loop has some responsibility among the call/execution stack, callback/task queue, and microtask queue. It simply checks the call stack if it is empty, if empty then it pulls tasks either from the microtask queue (priority queue) or task queue and pushes it into the call stack which eventually executes the function/callback and the callback gets **popped off** the stack after the execution is done and finally it returns.

**Concurrency model**: This is based on the event loop. Let’s think about the setTimeout API which runs inside the Web API block which is a part of the javascript runtime environment and finally the event loop moves forward the callback of the setTimeout API to the callback queue as soon as the setTimeout function execution is done (when the duration is over). Later on the event loop again moves it forward to the call stack to be executed. We are seeing that when the the program pointer goes to the setTimeout() function it didn’t execute that instead it handed over to the Web API as this is part of it and its declaration is over there and call stack started executing another task, so technically Javascript is performing multiple task at the same time though it’s a single threaded scripting programming language.

To try it online here are some links below

[https://www.jsv9000.app](https://www.jsv9000.app/)

<http://latentflip.com/loupe>

## What is implicit and explicit binding?

If we pass a new **context** into a function using either ***call, apply* or *bind*** method then that function will point its owner to that particular context/object. This process is called explicit binding/indirect invocation other than that by default it’s implicit binding.

## The Difference Between Regular Functions and Arrow Functions

Regular functions created using function declarations or expressions that are constructible as well as callable. However, the arrow functions are only callable but not constructible.

For more information, arrow functions can never be used as constructor functions. Hence, they can never be invoked with the new keyword. It does not have its own context and always refers to the out context but regular function’s context is **dynamic** based on invocation.

### `this` keyword

**Regular functoin**: Inside of a regular JavaScript function, **this** value (aka the execution context) is dynamic.

The dynamic context means that the value of **this** depends on how the function is invoked. In JavaScript, there are **4 ways you can invoke a regular function**.

1. During a **simple invocation** the value of `this` equals to the **global object** (or undefined if the function runs in strict mode).

2. During a **method invocation** the value of `this` is the object owning the method.

3. During an **indirect invocation** using myFunc.call(thisVal, arg1, ..., argN) or myFunc.apply(thisVal, [arg1, ..., argN]) the value of `this` equals to the first argument.

For example:

const person = {

name: 'Bablu',

dob: '1991.11.10',

}

const person2 = {

name: 'Rahim',

dob: '1992.11.10',

}

const person3 = {

calculateAge (todayYear) {

const dob = new Date(this.dob).getFullYear();

todayYear = new Date(todayYear).getFullYear();

return todayYear - dob;

}

}

person3.calculateAge.call(person, new Date()); // 31

function calculateAge (todayYear) {

const dob = new Date(this.dob).getFullYear();

todayYear = new Date(todayYear).getFullYear();

return todayYear - dob;

}

calculateAge.call(person, new Date()); // 31

4. During a **constructor invocation** using new keyword `this` equals to the newly created instance.

**Arrow Function**: The arrow function doesn't define its own execution context. In other words, the arrow function resolves this lexically. `**this**` value inside of an arrow function always equals to the `**this**` value of the outer function.

=> `this` resolved lexically is one of the great features of arrow functions. As the arrow function doesn't define its own **this**; no more **const self = this** or **innerFn.bind(this)** workarounds.

=> The indirect invocation of an arrow function using **myArrowFunc.call(thisVal)** or **myArrowFunc.apply(thisVal)** doesn't change the value of `this`; the context value is always resolved lexically.

For Example:

let person = {

name: 'Bablul',

id: 1,

showTasks: function() {

const \_this = this;

return function showDetails() {

return `Name: ${\_this.name}`;

}

}

};

let person = {

name: 'Bablu’',

id: 1,

showTasks: function() {

return showDetails = () => {

return `Name: ${this.name}`;

}

}

};

### *arguments* object

Inside the body of a regular function, the *arguments* is a special array-like object containing the list of arguments which the function has been invoked with. On the other side, no **arguments** special keyword is defined inside an arrow function. But if it's an inner function then the arguments object of the outer function can be accessed through it even using the rest parameter.

### Implicit return

If the return statement is missing, or there's no expression after the return statement, the regular function implicitly returns **undefined**. But the inline arrow function is implicitly returned without the use of *return* keyword.

### Methods on classes

The regular functions are the usual way to define methods on classes. Sometimes you'd need to supply the method as a callback, for example to setTimeout() or an event listener. In such cases, you might encounter difficulties accessing `this` value. It happens when the method is separated/extracted from the object at that time the method loses its context and is converted into a new function and then its context becomes the global/window object.

Let's bind `this` value manually to the right context:

setTimeout(obj.logName.bind(obj), 1000);

So obj.logName.bind(obj) binds `this` value to the obj instance. Now you're sure that the method doesn't lose the context.

You can use the arrow function as methods inside classes. You can use a method as a callback without any manual binding of `this`.

Now, in contrast with regular functions, the method defined using an arrow binds `this` lexically to the class instance.

## Blocking & Non-blocking

Javascript is also known as a single threaded programming language at the same time event driven programming language.

**Blocking**: It flows stack data structure (last in first out) and executes events synchronously.

**Non-blocking**: It flows queue data structure (first in first out) and executes events asynchronously.

## What is hoisting?

Hoisting is a mechanism where variables and function declarations are moved to the top of their scope before executing any code. In JavaScript, all declarations (functions, variables, or classes) are "hoisted" to the top of their scope. In other words, they are parsed or evaluated before any other code is executed. This is opposed to expressions, which are evaluated inline. All expressions in JavaScript are initially set to **undefined**.

So it just gives us an advantage that no matter where **functions and variables are declared**. It says **undefined** value instead of **reference error** for expressions and for declarations they work as usual.

So before beginning any code execution, the global execution context (GEC) scans function declarations and makes them **available** for us and also scans expressions and makes them **undefined** for us.

**N.B**: Technically **let** and **const** are not hoisted (they are just hoisted in temporal dead zone)

## What is context?

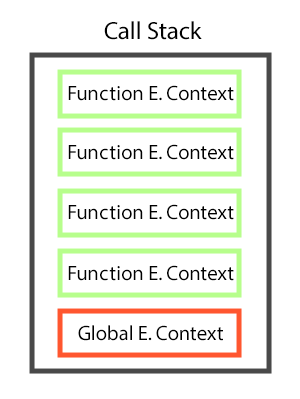
In JavaScript, “context” refers to an object. Within the object, the keyword “this” refers to that object (i.e. “self”). When a method is executed the keyword “this” refers to the object that the function is executed in.

## What is execution context?

Execution context (EC) is defined as the environment in which the JavaScript code is executed. By environment, JavaScript engine has access to code at a particular time.

Normally execution context in JavaScript is of two types:

1. **Global execution context (GEC)**: This is the default execution context in which JS code starts its execution when the file first loads in the browser. All of the global code i.e. code which is not inside any function or object is executed under the global execution context. GEC cannot be more than one because only one global environment is possible for JS code execution as the JS engine is single threaded.
2. **Functional execution context (FEC)**: Functional execution context is defined as the context created by the JS engine whenever it finds any function call. Each function has its own execution context. It can be more than one. While executing the global execution context code, if the JS engine finds a function call, it creates a new functional execution context for that function. In the browser context, if the code is executing in strict mode, then the value of **this** is **undefined** else it is the **window** object in the function execution context.

**Execution context stack (ECS) or Call stack**: 

Execution context stack is a stack data structure, i.e. last in first out data structure, to store all the execution stacks created during the life cycle of the script. Global execution context is present by default in the execution context stack and it is at the bottom of the stack. While executing the global execution context code, if JS engines find a function call, it creates a functional execution context for that function and pushes it on top of the execution context stack. JS engine executes the function whose execution context is at the top of the execution context stack. Once all the code of the function is executed, JS engines pop out that function’s execution context and then again starts executing the next one and so on until all functions are executed.

## What is lexical scope in JavaScript?

A lexical scope (sometimes known as static scope) in JavaScript means that the inner (nested) functions have access to the scope of their outer functions. But the opposite is not true; the variables defined inside a function will not be accessible from outside of that function. **This helps to make private variables because even after the outer function execution complete its variables are available for the inner function**.

**Note**: The **const** and **let** are any block scoped variables but **var** is only function scoped like we can access it from outside of the for loop though which was defined inside it.

For example:

{

var abc = 'abc';

let abc2 = 'abc2';

console.log(`Result: ${abc} and ${abc2}`); // Result: abc and abc2

}

console.log(`From outside of abc: ${abc}`); // From outside of abc: abc

console.log(`From outside of abc2: ${abc2}`); // Uncaught ReferenceError: abc2 is

not defined

## What is Closure in JavaScript?

A closure is a function which has its lexical scope. So a closure captures variables from its lexical scope. In other words, A closure is a **feature** in JavaScript where an inner function has access to the outer (enclosing) function's variables where it was initially created, even if it was being executed outside of it.

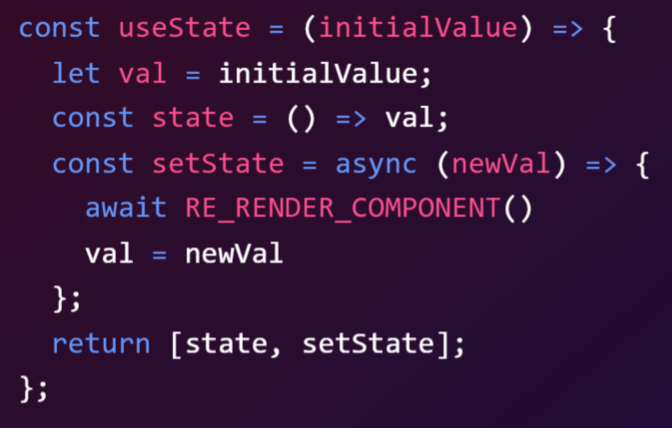
**The closure has three scope chains:**

- It has access to its own scope, i.e: variables defined between its curly brackets.

- It has access to the outer function's variables.

- It has access to the global variables.

For example:



In this case, **state** and **setState** are functions that are being executed outside of their scope, but they can still access the scope where they were initially declared inside. So **Closures** give you the power to create functions with a “memory” that persist, which means you execute the function again it will have **reference** to the previous execution. Let’s see another example:

const playGame = () => {

let counter = 0;

const increment = () => {

if(counter === 3){

console.log("Don't you have something better to do?")

return

}

counter+=1

console.log(counter)

}

return increment;

}

const onClick = playGame();

onClick(); // 1

onClick(); // 2

onClick(); // 3

onClick(); // Don't you have something better to do?

Exactly, closure keeps a “cache” or a “memory” of the function’s scope that the function can access even if it is being executed outside.

## Promises in JavaScript?

A promise is an object that may produce a single value in the future; either a resolved value, or a reason (an error message) that is not resolved (e.g., a network error occurred).

How can the Promise class help us?

It is now the standard way to deal with asynchronous code. Your code will actually be faster because you can perform other tasks while waiting for an asynchronous function to finish running. On top of that, your user interface won't freeze anymore and the users will think that your code is faster.

First of all, to use the Promise class, all you have to do is this:

// create

const promise = new Promise(function(resolve, reject) {

if(true) {

resolve('Some data'); // success

} else {

reject('An error message'); // error

}

});

// consume

promise

.then((data) => console.log(data))

.catch((err) => console.error(err));

Chaining the promises: Chaining promises is a proper way to tell JavaScript the next thing to do after an asynchronous task is done. It's easier to explain with an example so take a look at the following code and run it. For example:

doSomething1()

.then(doSomething2)

.then(doSomething3)

.catch(err => console.error(err))

Example 2

const promise = job1();

promise

.then(function(data1) {

console.log('data1', data1);

return job2();

})

.then(function(data2) {

console.log('data2', data2);

return 'Hello world';

})

.then(function(data3) {

console.log('data3', data3);

});

function job1() {

return new Promise(function(resolve, reject) {

setTimeout(function() {

resolve('result of job 1');

}, 1000);

});

}

function job2() {

return new Promise(function(resolve, reject) {

setTimeout(function() {

resolve('result of job 2');

}, 1000);

});

}

Ans:

data1 result of job 1

data2 result of job 2

data3 Hello world

## Dynamically adding properties to objects

A JavaScript object is just a collection of **key-value pairs** called properties. Once you have created an object, you might want to read or change them. Object properties can be accessed by using **the** **dot notation or the bracket notation.**

Oftentimes, there are variables in your code that you would like to put into an object. You will see code like this:

let a = 'foo',

b = 42,

c = {};

let o = {

a: a,

b: b,

c: c

}

With ECMAScript 2015, there is a shorter notation available to achieve the same:

// Shorthand property names (ES2015)

let o = {a, b, c}

A property of an object can also refer to a function or a getter or setter method.

In ECMAScript 2015, a shorthand notation is available, so that the keyword **function** is no longer necessary.

// Shorthand method names (ES2015)

let o = {

method(parameters) {},

}

let o = {

\*generator() {

...........

}

};

### Computed property names/Variable as a property

Starting with ECMAScript 2015 (ES6), the object initializer syntax also supports computed property names. That allows you to put an expression in brackets [], that will be computed and used as the property name. It’s gonna make our application so fast by removing lots of ternary or conditional operations.

// Computed property names (ES2015)

const name = "userName";

const obj = {[name]: "Bablu Ahmed", id: 12}; //{userName: "Bablu Ahmed", id: 12}

let i = 0

let a = {

['foo' + ++i]: i,

['foo' + ++i]: i,

['foo' + ++i]: i

}

console.log(a);

Output: {foo1: 1, foo2: 2, foo3: 3}

let param = 'size'

let config = {

[param]: 12,

['mobile' + param.charAt(0).toUpperCase() + param.slice(1)]: 4

}

console.log(config) // {size: 12, mobileSize: 4}

### Spread properties

Merging objects is now possible using a shorter syntax than Object.assign().

let obj1 = { foo: 'bar', x: 42 }

let obj2 = { foo: 'baz', y: 13 }

let clonedObj = { ...obj1 }

// Object { foo: "bar", x: 42 }

let mergedObj = { ...obj1, ...obj2 }

// Object { foo: "baz", x: 42, y: 13 }

## Prototypes

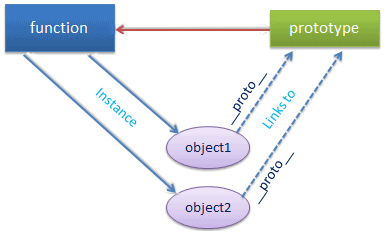
JavaScript is a prototype based language (not class based like Java, C++, etc. even though from ES6 the class feature has been added behind the scene it also uses prototypes). All objects in JavaScript are instances of **Object**. When an object is created that inherits properties and methods from **Object.prototype**. For example, whenever we create a function in JavaScript, the JavaScript engine adds a **prototype** property to the function. The **prototype** property is basically an object where we can attach methods and properties which will be shareable across all the instances of that constructor function.

There are some ways to use prototypes in JavaScript. Let's see one by one.

### #1 Using a constructor

We know in JavaScript, almost "everything" is objects. So a function is also an object and all functions have a special property named **prototype** which is an object. When you call a function as a constructor and create an object/instance, the **prototype** property is set as the **\_\_proto\_\_** property for the newly constructed object. This is something like

**FunctionName.prototype === object1.\_\_proto\_\_** returns **true** as illustrated below:



**Example 1**

const personPrototype = {

greet() {

console.log(`Hello, my name is ${this.name}!`);

}

}

function Person(name) {

this.name = name;

}

Person.prototype = personPrototype;

const obj1 = new Person('Bablu');

obj1.greet(); // Hello, my name is Bablu!

As you can see our **Person function** inherits properties and methods of the **personPrototype** object.

**Example 2**

function Person(name, profession) {

this.name = name;

this.profession = profession;

}

// setting a prototype of Person constructor

Person.prototype.org = 'W3 Public';

const personObj = new Person('Bablu', 'Teacher');

personObj.email = 'bablukpik@gmail.com';

function Teacher(dept) {

this.dept = dept;

}

// setting a prototype of the Teacher constructor

// with the instance of the Person constructor

Teacher.prototype = personObj;

const teacherObj = new Teacher('CSE');

console.log(teacherObj); // { dept: 'CSE' }

console.log(teacherObj.name); // Bablu

console.log(teacherObj.org); // W3 public

console.log(teacherObj.dept); // CSE

console.log(teacherObj.email); // [bablukpik@gmail.com](mailto:bablukpik@gmail.com)

**Example 3**

const personPrototype = {

getInfo() {

return `Name: ${this.name}, Age: ${this.age}`;

}

};

function Person(name, age) {

this.name = name;

this.age = age;

};

// setting ptototype value which is by default empty

Person.prototype = personPrototype;

function Teacher(name, age, subject) {

Person.call(this, name, age);

this.subject = subject;

}

const teacherObj = new Teacher(

'Bablu',

30,

'JavaScript'

);

// console.log(teacherObj.getInfo()); // TypeError: teacherObj.getInfo is not a function

console.log(teacherObj.name); // Bablu

console.log(teacherObj.subject); // JavaScript

### #2 Using Object.create

**Object.create()** method is used to create a new object with the specified prototype object. The applications of this method are used for implementing inheritance.

**Syntax**: Object.create(prototype[, propertiesObject])

**prototype** : It is the prototype object from which a new object has to be created.

**propertiesObject** : It is an optional parameter. If you want to specify properties and methods to the newly created object during creation of the instance.

const personPrototype = {

getInfo() {

return `Name: ${this.name}, Age: ${this.age}`;

}

};

function Person(name, age) {

this.name = name;

this.age = age;

};

// setting ptototype value which is by default empty

Person.prototype = personPrototype;

function Teacher(name, age, subject) {

Person.call(this, name, age);

this.subject = subject;

}

// setting ptototype value which is by default empty

Teacher.prototype = Object.create(Person.prototype);

const teacherObj = new Teacher(

'Bablu',

30,

'JavaScript'

);

console.log(teacherObj.getInfo()); // Name: Bablu, Age: 30

console.log(teacherObj.name); // Bablu

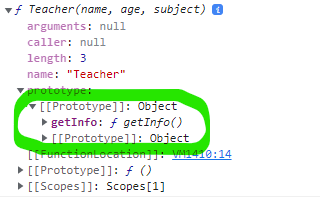
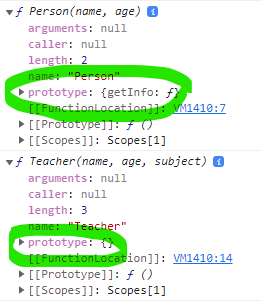
console.log(teacherObj.subject); // JavaScript

Here the statement **Teacher.prototype = Object.create(Person.prototype**); we can modify it instead of using Object.create method like so

Teacher.prototype = Person.prototype; this similar to this Teacher.prototype = personPrototype;

But this is duplicate and it will have performance issues because we’re assigning the same personPrototype to every constructor. If we use Objecte.create method then the newly created object is gonna point to the Person’s prototype. See the difference in the following figures

Fig. (1) Fig. (2)



In the figure 1 we’re seeing Teacher’s prototype is empty but as this is also an object it also has a prototype property which is actually pointing to the parent prototype in the figure 2 with getInfo() function. We’re also seeing another prototype property right below the getInfo() function in the figure 2 which is pointing to the master Object.

**Example 2**

function Person() {}

Person.prototype.personProp = 'Person val';

function Teacher() {}

const personProto = Object.create(Person.prototype);

personProto.teacherProp = 'Teacher val';

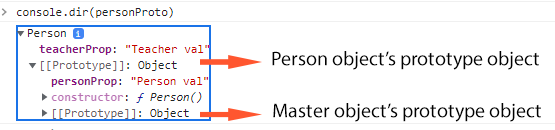
Teacher.prototype = personProto;

const teacherObj = new Teacher();

console.log(teacherObj.personProp); // Person val

console.log(teacherObj.teacherProp); // Teacher val

We know every object has a built-in prototype property. So here the personProto instance is created based on the Person constructor prototype. So the personProto instance’s prototype will point to Person's constructor prototype. Let’s see in the following figure.



You can also access the prototype value of an object like this Object.getPrototypeOf(personProto). If you expand the constructor as we’re seeing in the figure you’d be able to see the details of it. Finally you’re also seeing the master object’s prototype object, if you console **Object.prototype** then what you’re gonna see is similar to this.

### #3 Object.setPrototypeOf

**Example 1**

const user = { name: 'John' };

const arr = [1, 2, 3];

console.log('Original state');

console.log(user); // { name: 'John' }

console.log(user[1]); // undefined

console.log(user.\_\_proto\_\_); // {}

console.log(user.length); // undefined

Object.setPrototypeOf(user, arr); // add the arr to user’s prototype

console.log('Modified state');

console.log(user); // Array { name: 'John' }

console.log(user[1]); // 2

console.log(user.\_\_proto\_\_); // [ 1, 2, 3 ]

console.log(user.length); // 3

**Example 2**

function Person() { }

Person.prototype.personProp = 'Person val';

function Teacher() { }

const teacherProto = { teacherProp: 'Teacher val' };

Object.setPrototypeOf(teacherProto, Person.prototype); // add Person prototype to teacherProto

Teacher.prototype = teacherProto; // add teacherProto to the Teacher prototype

const teacherObj = new Teacher();

console.log(teacherObj.personProp);

console.log(teacherObj.teacherProp);

## #4 Setting the \_\_proto\_\_ property

**Example 1**

function Person() {}

Person.prototype.personProp = 'Person val';

function Teacher() {}

const proto = {

teacherProp: 'Teacher val',

\_\_proto\_\_: Person.prototype

};

Teacher.prototype = proto;

const inst = new Teacher();

console.log(inst.personProp);

console.log(inst.teacherProp);

**Example 2**

const teacherObj = {

name: 'Bablu',

\_\_proto\_\_: {

personProp: 'Person val',

\_\_proto\_\_: {

teacherProp: 'Teacher val',

\_\_proto\_\_: Object.prototype

}

}

};

console.log(teacherObj.name);

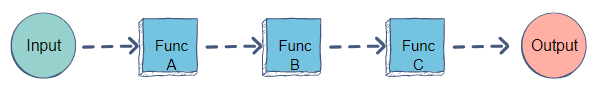
console.log(teacherObj.teacherProp);

console.log(teacherObj.personProp);

## Composition Vs Inheritance

### Composition

Function composition is an approach where the result of one function is passed onto the next function, which is passed to another until the final function is executed for the final result. In other words, **function composition is the process of combining two or more functions to produce a new function**. Function compositions can be composed of any number of functions.



**Example**

// traditional approach

const double = x => x \* 2;

const square = x => x \* x;

// variant one

var output1 = double(2);

var output2 = square(output1);

console.log(output2); // 16

// variant two

var output\_final = square(double(2));

console.log(output\_final); // 16

In the code above, we can see that we need to call the double function followed by the square function to square a term that has been doubled.

#### Function Composing and Piping

The modern approach of composition is to use the compose and pipe functions.

**Compose function**: The compose function takes any number of functions and invokes them all one after the other.

**Example 1**

// traditional way

const sum = (x, y) => x + y;

const x = 2;

const y = 2;

const result = sum(x, y);

console.log(result);

// composing functions

const sum = x => y => x + y;

const x = 2;

const y = 2;

// const sumX = sum(x);

// const sumY = sumX(y);

// console.log(sumY); // 4

// Or

const result = sum(x)(y);

console.log(result); // 4

**Example 2**

// function composition of any number of functions

const compose = (...fns) => x => fns.reduceRight((y, f) => f(y), x);

const double = x => x \* 2

const square = x => x \* x

// function composition

var output\_final = compose(square, double)(2);

console.log(output\_final);

**N.B**: The reduce() method starts at the first element and travels toward the last, whereas the reduceRight() method starts at the last element and travels backward the first.

**Pipe function**: We can reverse the order of the function invocation by using the pipe function process.

**Example**

// function composition using pipe of any number of functions

const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);

const double = x => x \* 2

const square = x => x \* x

// function pipe

var output\_final = pipe(square, double)(2);

console.log(output\_final); // 8

**N.B:** We can use Lodash’s flowRight (compose) and flow (pipe) functions.

### Inheritance

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more. Using class inheritance, a class can inherit all the methods and properties of another class. Inheritance is a useful feature that allows code reusability.

**Example**

// parent class animal

class Animal {

constructor(name, weight) {

this.name = name;

this.weight = weight;

}

eat() {

return `${this.name} is eating!`;

}

sleep() {

return `${this.name} is going to sleep!`;

}

wakeUp() {

return `${this.name} is waking up!`;

}

}

// sub class gorilla

class Gorilla extends Animal {

constructor(name, weight) {

super(name, weight);

}

climbTrees() {

return `${this.name} is climbing trees!`;

}

dailyRoutine() {

return `${this.climbTrees()} ${super.wakeUp()} ${super.eat()} ${super.sleep()}`;

}

}

function display(content) {

console.log(content);

}

const gorilla = new Gorilla('George', '160Kg');

display(gorilla.sleep());

display(gorilla.dailyRoutine());

// OUTPUT:

// George is going to sleep!

// George is climbing trees! George is waking up! George is eating! George is going to sleep!

*As javascript class doesn't support multiple inheritance so we can use factory functions for each functionality and call them in a final factory function which will return an aggregation object of all objects.*

function attackerAndWalker({ name }) {

return {

attack: () => console.log(`${name} attacked`),

walk: () => console.log(`${name} walked`)

}

}

function swimmer({ name }) {

return {

swim: () => console.log(`${name} swam`)

}

}

function flier({ name }) {

return {

fly: () => console.log(`${name} flew`)

}

}

function swimmingMonsterCreator(monster) {

return {

...monster,

...attackerAndWalker(monster),

...swimmer(monster)

}

}

function flyingSwimmingMonsterCreator(monster) {

return {

...monster,

...attackerAndWalker(monster),

...flier(monster),

...swimmer(monster),

}

}

const obj = flyingSwimmingMonsterCreator({ name: 'Duck' });

console.log(obj.name); // Duck

obj.attack(); // Duck attacked

obj.walk(); // Duck walked

obj.swim(); // Duck swam

obj.fly(); // Duck flew

## What is aggregation in JavaScript?

An aggregate is an object which contains other objects. The ability to combine several objects into a new one is known as aggregation. Multiple small objects are easier to manage than one large object.

**For example**

const obj1 = {};

const obj2 = {};

const obj3 = {};

const aggs = {...obj1, …obj2, …obj3};

## Shallow and deep copy in JavaScript

**Shallow Copy**: As a reference type variable receives is an address or the location instead of a value so 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 other words, 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 variables 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. For example:

const employee = {

id: "102",

name: "Bablu",

address: "Dhaka, Bangladesh",

salary: 100000

}

const newEmployee = employee; // Shallow copy

// After modification

newEmployee.name = "Faruk";

console.log(employee.name) // Faruk

**Deep Copy**: Deep copy makes a copy of all the members of the old object, allocates a separate memory location for the new object and then assigns the copied members to the new object. For example:

const employee = {

id: "102",

name: "Bablu",

address: "Dhaka, Bangladesh",

salary: 100000

}

const newEmployee = JSON.parse(JSON.stringify(employee)); // Deep copy

// After modification

newEmployee.name = "Faruk";

console.log(employee.name); // Bablu

console.log(newEmployee.name); // Faruk

// Some ways of deep copying

1. const arr2 = JSON.parse(JSON.stringify(arr1)); // an object as well

2. const arr2 = arr1.slice(0);

3. const arr2 = [].concat(arr1); // concat with an empty array

4. const arr2 = Array.from(arr1);

5. const obj2 = Object.create({}, obj1);

6. const obj2 = Object.assign({}, obj1);

7. Using the ES6 Spread Operator (...)

## JavaScript Scope

Scope determines the visibility and accessibility or availability of a variable in certain parts of the code. We know that during the **creation phase** of the global execution context, the JavaScript engine performs the following tasks:

1. Create the global object i.e., **window** in the web browser or **global** in Node.js.
2. Create the **this** object and bind it to the **global object**.
3. Setup a memory heap for storing **variables** and **function references**.
4. Store the **variables with the initial values as undefined** and **function declarations/function references** in the memory heap.

After the creation phase, the global execution context moves to the **execution phase**. During the **execution phase**, the JavaScript engine executes the code line by line, assigns the values to variables, and executes the function calls.

For each function call, the JavaScript engine creates a new **function execution context**. To keep track of all the **execution contexts**, including the **global execution context** and **function execution contexts**, the JavaScript engine uses the **call stack**.

JavaScript has **three** scopes:

1. Global scope
2. Local/Function scope
3. Block scope (started from ES6)

### Global Scope

When the JavaScript engine executes a script after first loading, it creates a global execution context. It also assigns variables that you declare outside of functions to **the** **global execution context**. These variables are in the global scope also known as global variables.

Let's see an example of a global scope variable.

let a = "hello";

function greet () {

console.log(a);

}

greet(); // hello

In the above program, variable **a** is declared at the top of a program and is a global variable. It means the variable **a** can be used anywhere in the program.

The value of a global variable can be changed inside a function. For example,

let a = "hello";

function greet() {

a = 3;

}

// before the function call

console.log(a); // hello

// after the function call

greet();

console.log(a); // 3

**Note**: It is a good practice to avoid using global variables because the value of a global variable can change in different areas in the program. It can introduce unknown results in the program.

In JavaScript, a variable can also be used without declaring it. If a variable is used without declaring it, that variable automatically becomes a global variable.

For example,

function greet() {

a = "hello"

}

greet();

console.log(a); // hello

In the above program, variable **a** is a global variable.

**Note**: In JavaScript, there is "strict mode"; in which a variable cannot be used without declaring it.

### Local Scope

We know that each function is born with a new scope. The variables that you declare inside a function or its scope are called local variables. Variables declared with **var**, **let** and **const** are quite similar when declared inside a function.

### Block scope

ES6 provides the **let** and **const** keywords that allow you to declare variables in block scope.

Generally, whenever you see curly brackets {}, it is a block. It can be the area within the if, else, switch conditions or for, do while, and while loops. For example:

// global variable

let a = 'Hello';

function greet() {

// local variable

let b = 'World';

console.log(a + ' ' + b);

if (b == 'World') {

// block-scoped variable

let c = 'hello';

console.log(a + ' ' + b + ' ' + c);

}

// variable c cannot be accessed here

console.log(a + ' ' + b + ' ' + c); // error

}

greet();

In the above program,

* a is a global variable. It can be accessed anywhere in the program.
* b is a local variable. It can be accessed only inside the function greet.
* c is a block-scoped variable. It can be accessed only inside the if statement block.

In JavaScript, **var** is only function scoped, and other two the **let** and **const** keywords are block-scoped (variable can be accessed only in the immediate block i.e: {...}).

### Scope chain

Consider the following example:

var message = 'Hi';

function say() {

console.log(message);

}

say(); // Hi

In this example, we reference the variable **message** inside the **say()** function. Behind the scenes, JavaScript performs the following:

Look up the variable **message** in the current context (function execution context) of the **say()** function. If it cannot find any then try to find the variable **message** in the outer execution context which is the global execution context. It finds the variable **message**. In case it doesn’t find in the global execution context then try again searching in the prototype chain if it doesn’t find the **message** variable then finally throws a **reference error**.

The way that JavaScript resolves a variable is by looking at it in its current scope, if it cannot find the variable, it goes up to the outer scope, and then prototype chain which is called the scope chain.

We know that the **Lexical scope** is only at the function level and Lexical Scoping defines how variable names are resolved in nested functions. The inner functions contain the scope of parent functions even if the parent function has returned but if there is not defined that variable inside the parent function then the inner function uses the **Scope chain**.

## Getter & Setter (Accessors) in JavaScript

In JavaScript, there are two kinds of object properties:

1. Data properties
2. Accessor properties

### Data Property

Here's an example of data property.

const student = {

// data property

firstName: 'Bablu';

};

### Accessor Property

In JavaScript, accessor properties are methods that get or set the value of an object. For that, we use these two keywords:

* get - to define a getter method to get the property value
* set - to define a setter method to set the property value

### JavaScript Getter

In JavaScript, getter methods are used to access the properties of an object. For example,

const student = {

// data property

firstName: 'Bablu',

// accessor property(getter)

get getName() {

return this.firstName;

}

};

// accessing data property

console.log(student.firstName); // Bablu

// accessing getter methods

console.log(student.getName); // Bablu

In the above program, when accessing the value, we access the value as a property.

### JavaScript Setter

In JavaScript, setter methods are used to change the values of an object. For example,

const student = {

firstName: 'Bablu',

// accessor property(setter)

get getName() {

return this.firstName;

},

// accessor property(setter)

set changeName(newName) {

this.firstName = newName;

}

};

console.log(student.firstName); // Bablu

// change the value of the firstName property using a setter

student.changeName = 'Faruk';

// accessing data property

console.log(student.firstName); // Faruk

// accessing getter methods

console.log(student.getName); // Faruk

### Getters and Setters using Class

The above code is reformatted below using a Class.

class Student {

constructor(firstName) {

this.firstName = firstName;

}

//accessor property(setter)

get getName() {

return this.firstName;

}

//accessor property(setter)

set changeName(newName) {

this.firstName = newName;

}

}

const student1 = new Student('Bablu');

console.log(student1.firstName); // Bablu

// change the value of the firstName property using a setter

student1.changeName = 'Faruk';

console.log(student1.firstName); // Faruk

console.log(student1.getName); // Faruk

## Pure Functions

In Functional Programming, a pure function is a function which:

1. Is 𝗱𝗲𝘁𝗲𝗿𝗺𝗶𝗻𝗶𝘀𝘁𝗶𝗰: Given the same input, always returns the same output

2. Has 𝗻𝗼 𝘀𝗶𝗱𝗲-𝗲𝗳𝗳𝗲𝗰𝘁𝘀, such as

- Reading/mutating the shared state to process the output

- Mutating your input

- I/O operations (console.log)

- HTTP calls (AJAX/fetch)

- Changing the filesystem

- Querying the DOM

By using Pure Functions code becomes:

- highly reusable

- more predictable

- much easier to test

- easier to reason about



## Currying

Currying means evaluating functions with multiple arguments and decomposing them into a sequence of functions with a single argument. So, instead of taking all arguments at once, the function takes the first argument and returns a new function, which takes second argument and returns a new function, which takes the third… and so on until all arguments are provided and the final function is executed.

For example:

const buildSammy = (ingred1, ingred2, ingred3) =>

`${ingred1}, ${ingred2}, ${ingred3}`;

We could rewrite the above function as follows that’d be for each argument one function sequentially

const buildSammy = ingred1 => ingred2 => ingred3 =>

`${ingred1}, ${ingred2}, ${ingred3}`;

const mySammy = buildSammy("turkey")("cheese")("bread");

console.log('mySammy:', mySammy); // mySammy: turkey, cheese, bread

Example 02:

const multiply = (x, y) => x \* y;

const curriedMultiply = x => y => x \* y;

console.log(‘curriedMultiply’, curriedMultiply(3)(3)); // 9

Another common use of currying is function composition. There is a full section of function composition which is **Composition Vs Inheritance**. The function composition can be composed of any number of functions.

For example:

// Compose function

const compose = (...fns) => x => fns.reduce((y, f) => f(y), x);

// Functions

const addFee = amount => amount + 2;

const addDiscount = amount => amount - 5;

// function composition

const grandTotal = compose(addFee, addDiscount)(100);

In the above code it’ll first add fee then add discount to the total amount and then finally returns the composed/grand total amount.

## What is the Design Pattern in JavaScript?

Design patterns are reusable solutions to commonly occurring problems in software design. They are proven solutions, easily reusable and expressive. They lower the size of your codebase, prevent future refactoring, and make your code easier to understand by other developers.

### Design Pattern Categorization

Design patterns can be categorized in multiple ways, but the most popular one is the following:

* Creational design patterns
* Structural design patterns
* Behavioral design patterns
* Concurrency design patterns
* Architectural design patterns

Creational Design Patterns

These patterns deal with object creation mechanisms which optimize object creation compared to a basic approach.

Some of the popular design patterns in this category are:

* Factory method
* Abstract factory
* Builder
* Prototype
* Singleton

## What are the S.O.L.I.D Principles?

S: Single Responsibility Principle

O: Open-Closed Principle.

L: Liskov-Substitution Principle.

I: Interface Segregation Principle.

D: Dependency Inversion Principle.

**Single Responsibility Principle**: A class should only have one job/responsibility.

**Open-Closed Principle**: Objects or entities should be open for extension, but closed for modification.

**Liskov-Substitution Principle**: Any subclass object should be substitutable for the superclass object from which it is derived.

**Interface Segregation Principle**: A client should never be forced to implement an interface, but the interface should be based on clients.

**Dependency Inversion Principle**: High-level modules should not import anything from low-level modules; they should both depend on abstractions.

𝗦𝗼𝗳𝘁𝘄𝗮𝗿𝗲 𝗔𝗿𝗰𝗵𝗶𝘁𝗲𝗰𝘁𝘂𝗿𝗲: Modern SOLID

The SOLID principles are a time-tested rubric for creating quality software.

But in a world of multi-paradigm programming and cloud computing, are they still valid?

Absolutely yes!

Daniel Orner restates each of the five SOLID principles to a more general statement that can apply to OO, FP, or multi-paradigm programming.

1. Single responsibility principle

Original definition: “There should never be more than one reason for a class to change.”

New definition: “Each module should do one thing and do it well.”

2. Open-closed principle

Original definition: “Software entities should be open for extension, but closed for modification.”

New definition: “You should be able to use and add to a module without rewriting it.”

3. Liskov substitution principle

Original definition: “If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program.”

New definition: You should be able to substitute one thing for another if those things are declared to behave the same way.

4. Interface segregation principle

Original definition: “Many client-specific interfaces are better than one general-purpose interface.”

New definition: “Don’t show your clients more than they need to see”.

5. Dependency inversion principle

Original & New definition: “Depend upon abstractions, not concretions.”

To restate “modern SOLID” one more time:

- Don’t surprise the people who read your code.

- Don’t surprise the people who use your code.

- Don’t overwhelm the people who read your code.

- Use sane boundaries for your code.

- Use the right level of coupling—keep things together that belong together, and keep them apart if they belong apart.

Others:

API Gateway

REST principles

## Recursion

Recursion is a process of calling itself. A function that calls itself is called a recursive function. To prevent infinite recursion, we can use the **if...else statement** where one branch makes the recursive call, and the other doesn't. The condition that stops a recursive function from calling itself is known as the **base case**. Typically, you will find the recursive functions in data structures like binary trees, graphs and algorithms such as binary search and quicksort.

function recurse() {

if(true) {

// stop calling itself

//...

} else {

recurse();

}

}

recurse();

**For example**: To console log numbers from 1 to 5

function log(num){

if(num > 5){

return;

}

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

log(num + 1);

}

log(1);

**Example 02**: To console log numbers in reverse order

function countDown(fromNumber) {

console.log(fromNumber);

let nextNumber = fromNumber - 1;

if (nextNumber > 0) {

countDown(nextNumber);

}

}

countDown(3); // 3, 2, 1

**Example 03**: To sum like 1+2+3…+n

function sum(n) {

if (n <= 1) {

return n;

}

return n + sum(n - 1);

}

sum(3); // 6

This is how it’ll work as follows

3 + sum(2) // 3 + 3 = 6

2 + sum(1) // 2 + 1 = 3, since **sum(1)** will return 1 from the base case

## Handling Key-Value Relationships with Maps

A Map is an associative data structure. It contains **key-value pairs** and offers **fast** access to values based on **keys**. This is an alternative to the JS Object. We can create a **map** **object** with the **Map constructor** and handle it as follows.

1. It creates a map

let map = new Map();

// OR

map = new Map([

['1', 1],

['2', 2],

['3', 3]

]);

2. It checks wheather the map contains a key

map.has('1'); // true

map.has('10'); // false

3. It gets access to values

map.get('1'); // 1

map.get('10'); // undefined

4. It adds a new key-value pair or overwrites the value of an existing key

map.set('4', 4);

5. It deletes a key-value pair from the map

map.delete('4');

map.size; // 3

6. It deletes all elements from the map

map.clear();

map.size; // 0

## Handling Unique Data With Sets

A Set is a data structure modeled after the set known from math. It is an alternative to using an **array** when you only want unique elements.

The methods used by Sets to search for, delete and insert items all have a time complexity of just **O(1)** that means the size of the data has virtually no bearing on the run-time of these methods!

We can create a **set** **object** with the **Set constructor** and handle it as follows.

1. It creates a set

let set = new Set();

// OR

set = new Set([1, 2, 3, 4]); // 1, 2, 3, 4

2. It checks if the Set contains a value

set.has(1);

3. It adds an unique element to the set

set

.add(1)

.add(5); // set is now 1, 2, 3, 4, 5

4. It deletes an element from the set

set.delete(1); // set is now 2, 3, 4, 5

set.length; // always 0!

set.size; // 4

5. It deletes all elements from the set

set.clear(); // set is now empty

6. The **values()** method returns a new Iterator object that contains the values for each element in the Set object in insertion order.

const set1 = new Set();

set1.add(42);

set1.add('forty two');

const iterator1 = set1.values();

console.log(iterator1.next().value);

// expected output: 42

console.log(iterator1.next().value);

// expected output: "forty two"

7. The **entries()** method returns a new Iterator object that contains an array of [value, value] for each element in the Set object, in insertion order.

For example:

const set1 = new Set();

set1.add(42);

set1.add('forty two');

const iterator1 = set1.entries();

for (const entry of iterator1) {

console.log(entry);

// expected output: [42, 42]

// expected output: ["forty two", "forty two"]

}

Example 02:

const mySet = new Set();

mySet.add('foobar');

mySet.add(1);

mySet.add('baz');

const setIter = mySet.entries();

console.log(setIter.next().value); // ["foobar", "foobar"]

console.log(setIter.next().value); // [1, 1]

console.log(setIter.next().value); // ["baz", "baz"]

Example 03: Deleting an item with compare to an array

let arr = [], set = new Set(), n = 1000000;

for (let i = 0; i < n; i++) {

arr.push(i);

set.add(i);

}

const deleteFromArr = (arr, item) => {

let index = arr.indexOf(item);

return index !== -1 && arr.splice(index, 1);

};

console.time('Array');

deleteFromArr(arr, n);

console.timeEnd('Array');

console.time('Set');

set.delete(n);

console.timeEnd('Set');

Array: 1.122ms

Set: 0.015ms

Example 04: Searching for an item

let result;

console.time('Array');

result = arr.indexOf(123123) !== -1;

console.timeEnd('Array');

console.time('Set');

result = set.has(123123);

console.timeEnd('Set');

Array: 0.173ms

Set: 0.023ms

Example 05: Adding an item

console.time('Array');

arr.push(n);

console.timeEnd('Array');

console.time('Set');

set.add(n);

console.timeEnd('Set');

Array: 0.018ms

Set: 0.003ms

The Set was 6.73 times faster

We can also use Set as an array like this **const arr = [..set]**;

~~Data types~~

~~Dynamic typing~~

~~Concurrency Model and Event Loop~~

~~What is implicit and explicit binding?~~

~~The Difference Between Regular Functions and Arrow Functions~~

~~Blocking & Non-blocking~~

~~What is hoisting?~~

~~What is context?~~

~~What is execution context?~~

~~What is lexical scope in JavaScript?~~

~~What is closure in JavaScript?~~

Operators:

Loops:

~~Dynamically adding properties to objects:~~

~~Variable scope:~~

Constructors:

~~Prototypes~~:

~~Composition Vs Inheritance~~

~~What is aggregation in javascript?~~

~~`this` and `new` keywords:~~

~~Promises in JavaScript:~~

~~Shallow and deep copy in JavaScript~~

~~Getters & Setters in JavaScript~~

Array functions

Static methods

JavaScript Higher Order Functions

ES6, ES7, ES8, ES9, ES10