**Map in JS**

A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.

**New array values created using previous array.**

Const num=[1,3,5,9]

Const newNum= number.map(function(value) --- > *Here function can take 3 parameters i.e. value, index, array.*

{

Return value\*2;

})

Console.log(num);

Console.log(newNum);

**How to Create a Map**

You can create a JavaScript Map by:

* Passing an Array to new Map()

const fruits = new Map([  
  ["apples", 500],  
  ["bananas", 300],  
  ["oranges", 200]  
]);

* Create a Map and use Map.set()

// Create a Map  
const fruits = new Map();  
  
// Set Map Values  
fruits.set("apples", 500);  
fruits.set("bananas", 300);  
fruits.set("oranges", 200);

Note: Set() method can also be used to change existing map values.

fruits.set("apples", 200);

**The get() Method**

The get() method gets the value of a key in a Map:

fruits.get("apples");

**Maps are Objects**

typeof returns object:

ex. typeof fruits;

instanceOf map returns true:

ex. fruits instanceof Map;

**JavaScript Objects vs Maps**

|  |  |
| --- | --- |
| **Object** | **Map** |
| Not directly iterable | Directly iterable |
| Do not have a size property | Have a size property |
| Keys must be Strings (or Symbols) | Keys can be any datatype |
| Keys are not well ordered | Keys are ordered by insertion |
| Have default keys | Do not have default keys |

**Object oriented programming in JS**

* Class

Classes are **blueprints** of an Object. A class can have many Objects because the class is a **template** while Objects are **instances** of the class or the concrete implementation.   
Before we move further into implementation, we should know unlike other Object-Oriented languages there are **no classes in JavaScript** we have only Object. To be more precise, JavaScript is a prototype-based Object-Oriented Language, which means it doesn’t have classes, rather it defines behaviors using a constructor function and then reuses it using the prototype.

* **// Defining class using es6**

class Vehicle {

    constructor(name, maker, engine) {

**this**.name = name;

**this**.maker = maker;

**this**.engine = engine;

    }

    getDetails() {

**return** (`The name of the bike is ${**this**.name}.`)

    }

}

// Making object with the help of the constructor

let bike1 = **new** Vehicle('Hayabusa', 'Suzuki', '1340cc');

let bike2 = **new** Vehicle('Ninja', 'Kawasaki', '998cc');

console.log(bike1.name);    // Hayabusa

console.log(bike2.maker);   // Kawasaki

console.log(bike1.getDetails());

* **// Defining class in a Traditional Way.**

**function** Vehicle(name, maker, engine) {

**this**.name = name,

**this**.maker = maker,

**this**.engine = engine

};

Vehicle.prototype.getDetails = **function** () {

    console.log('The name of the bike is ' + **this**.name);

}

let bike1 = **new** Vehicle('Hayabusa', 'Suzuki', '1340cc');

let bike2 = **new** Vehicle('Ninja', 'Kawasaki', '998cc');

console.log(bike1.name);

console.log(bike2.maker);

console.log(bike1.getDetails());

* Object

An Object is a **unique** entity that contains **properties** and **methods**. For example “a car” is a real-life Object, which has some characteristics like color, type, model, and horsepower and performs certain actions like driving. The characteristics of an Object are called Properties in Object-Oriented Programming and the actions are called methods. An Object is an **instance** of a class. Objects are everywhere in JavaScript, almost every element is an Object whether it is a function, array, or string.

**Note:** A Method in javascript is a property of an object whose value is a function.

The object can be created in two ways in JavaScript:

* **Object Literal**
* **Object Constructor**
* Encapsulation

The process of **wrapping properties and functions**within a **single unit** is known as encapsulation.

// Encapsulation example

class person {

    constructor(name, id) {

**this**.name = name;

**this**.id = id;

    }

    add\_Address(add) {

**this**.add = add;

    }

    getDetails() {

        console.log(`Name is ${**this**.name},

        Address is: ${**this**.add}`);

    }

}

let person1 = **new** person('Mukul', 21);

person1.add\_Address('Delhi');

person1.getDetails();

* Abstraction

Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

// Abstraction example

**function** person(fname, lname) {

    let firstname = fname;

    let lastname = lname;

    let getDetails\_noaccess = **function** () {

**return** (`First name is: ${firstname} Last

            name is: ${lastname}`);

    }

**this**.getDetails\_access = **function** () {

**return** (`First name is: ${firstname}, Last

            name is: ${lastname}`);

    }

}

let person1 = **new** person('Mukul', 'Latiyan');

console.log(person1.firstname);

console.log(person1.getDetails\_noaccess);

console.log(person1.getDetails\_access());

* Inheritance

It is a concept in which some properties and methods of an Object are being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Objects inherit Objects i.e. certain features (property and methods) of one object can be reused by other Objects.

// Inheritance example

class person {

    constructor(name) {

**this**.name = name;

    }

    // method to return the string

    toString() {

**return** (`Name of person: ${**this**.name}`);

    }

}

class student extends person {

    constructor(name, id) {

        // super keyword for calling the above

        // class constructor

**super**(name);

**this**.id = id;

    }

    toString() {

**return** (`${**super**.toString()},

        Student ID: ${**this**.id}`);

    }

}

let student1 = **new** student('Mukul', 22);

console.log(student1.toString());

* Polymorphism

Polymorphism is one of the core concepts of object-oriented programming languages. Polymorphism means the same function with different signatures is called many times. In real life, for example, a boy at the same time may be a student, a class monitor, etc. So a boy can perform different operations at the same time. Polymorphism can be achieved by method overriding and method overloading