More Modern JS Concepts

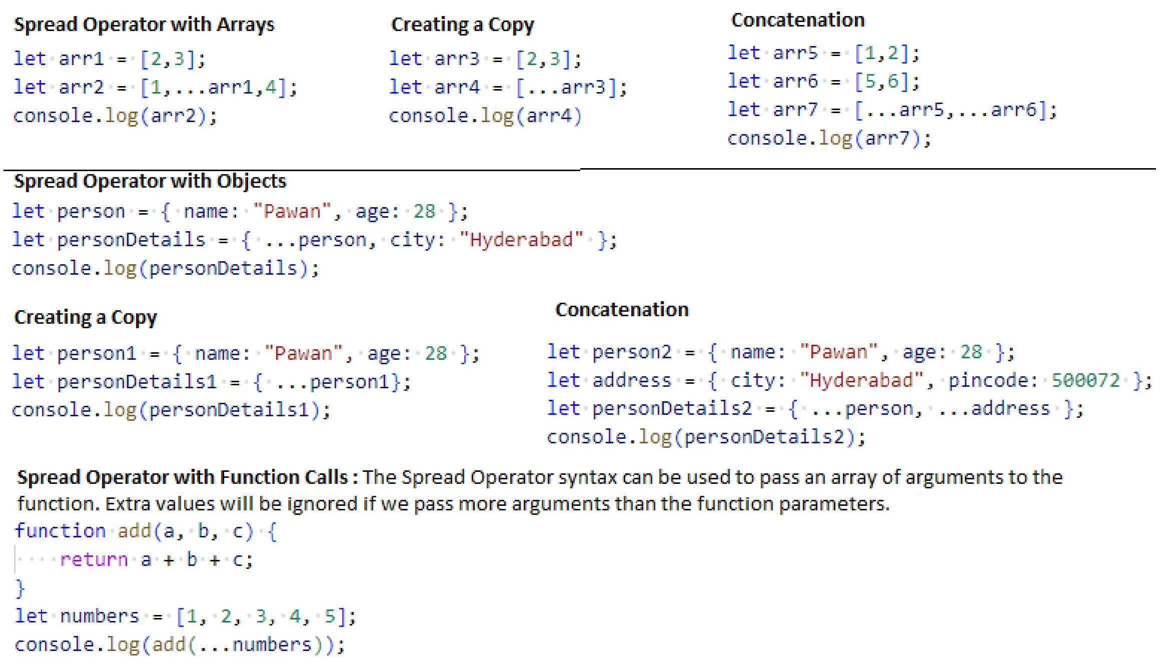
1. Spread Operator ¶

- The Spread Operator is used to unpack an iterable (e.g. an array, object, etc.) into individual elements.

In [1]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js69.png")
```

Out[1]:



Spread Operator with Arrays

```
let arr1 = [2,3];
let arr2 = [1,...arr1,4];
console.log(arr2);
```

Creating a Copy

```
let arr3 = [2,3];
let arr4 = [...arr3];
console.log(arr4)
```

Concatenation

```
let arr5 = [1,2];
let arr6 = [5,6];
let arr7 = [...arr5,...arr6];
console.log(arr7);
```

Spread Operator with Objects

```
let person = { name: "Pawan", age: 28 };
let personDetails = { ...person, city: "Hyderabad" };
console.log(personDetails);
```

Creating a Copy

```
let person1 = { name: "Pawan", age: 28 };
let personDetails1 = { ...person1};
console.log(personDetails1);
```

Concatenation

```
let person2 = { name: "Pawan", age: 28 };
let address = { city: "Hyderabad", pincode: 500072 };
let personDetails2 = { ...person, ...address };
console.log(personDetails2);
```

Spread Operator with Function Calls : The Spread Operator syntax can be used to pass an array of arguments to the function. Extra values will be ignored if we pass more arguments than the function parameters.

```
function add(a, b, c) {
    return a + b + c;
}
let numbers = [1, 2, 3, 4, 5];
console.log(add(...numbers));
```

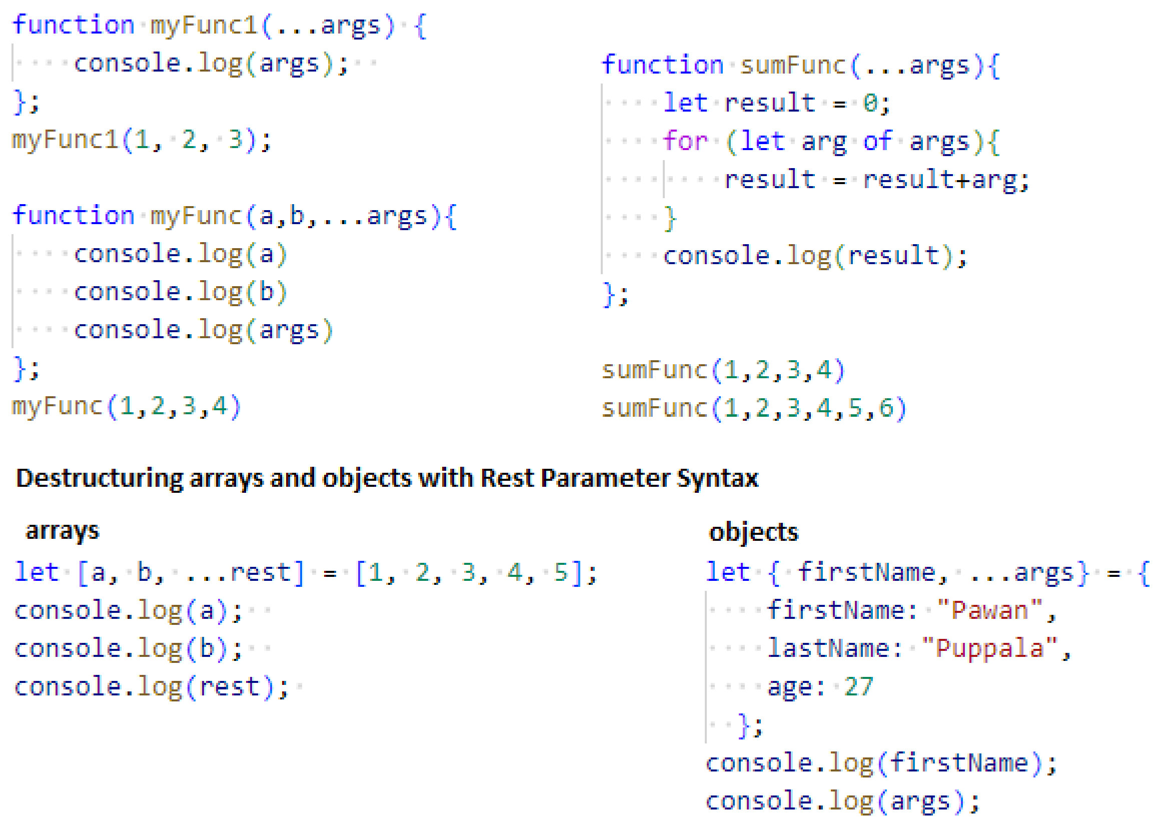
2. Rest Parameter

- With Rest Parameter, we can pack multiple values into an array.

In [4]:

```python
from IPython.display import Image
Image("E:/code/frontend/img/js70.png")
```

Out[4]:



```javascript
function myFunc1(...args) {
    console.log(args);
};
myFunc1(1, 2, 3);

function myFunc(a,b,...args){
    console.log(a)
    console.log(b)
    console.log(args)
};
myFunc(1,2,3,4)
```

```javascript
function sumFunc(...args){
    let result = 0;
    for (let arg of args){
        result = result+arg;
    }
    console.log(result);
};

sumFunc(1,2,3,4)
sumFunc(1,2,3,4,5,6)
```

Destructuring arrays and objects with Rest Parameter Syntax

arrays

```javascript
let [a, b, ...rest] = [1, 2, 3, 4, 5];
console.log(a);
console.log(b);
console.log(rest);
```

objects

```javascript
let { firstName, ...args} = {
    firstName: "Pawan",
    lastName: "Puppala",
    age: 27
  };
console.log(firstName);
console.log(args);
```

3. Functions

3.1 Default Parameters

- The Default Parameters allow us to give default values to function parameters.

function defaultFunc(a = 2, b = 5) { console.log(a);
console.log(b);
}

defaultFunc(3);

4. Template Literals (Template Strings)

- The Template Literals are enclosed by the backticks.
- They are used to:
  1. Embed variables or expressions in the strings
  2. Write multiline strings
- We can include the variables or expressions using a dollar sign with curly braces ${ }.

let Name = "Pawan"; console.log(`Hello ${Name}!`);

More Modern JS Concepts Part 2

1. Operators

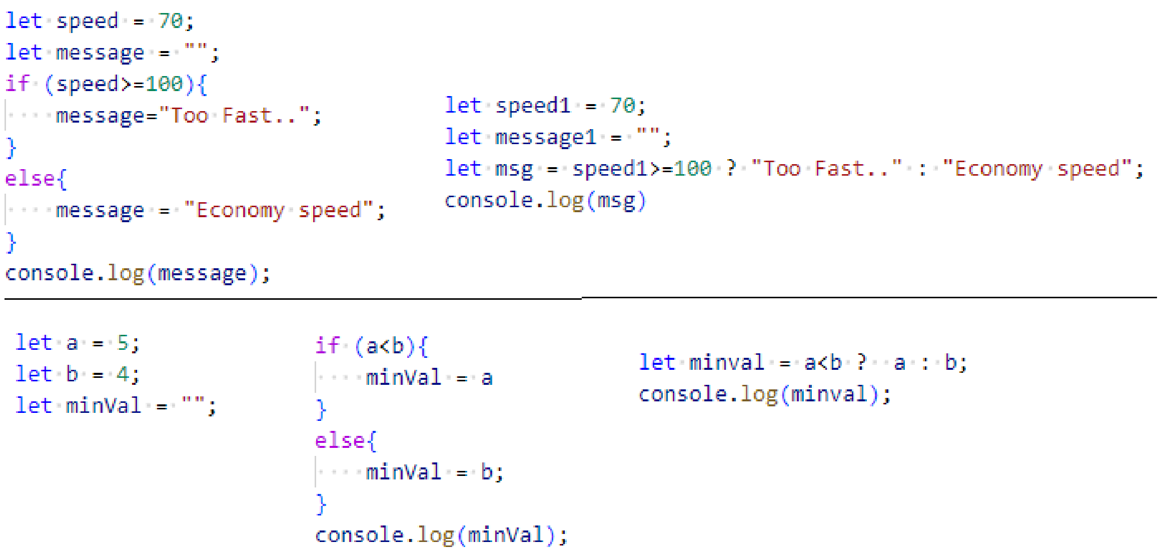
1.1 Ternary Operator

- A Ternary Operator can be used to replace if...else statements in some situations.
- Syntax: condition ? expressionIfTrue : expressionIfFalse

In [8]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js71.png")
```

Out[8]:



```
let speed = 70;
let message = "";
if (speed>=100){
    message="Too Fast..";
}
else{
    message = "Economy speed";
}
console.log(message);
```

```
let speed1 = 70;
let message1 = "";
let msg = speed1>=100 ? "Too Fast.." : "Economy speed";
console.log(msg)
```

```
let a = 5;
let b = 4;
let minVal = "";
```

```
if (a<b){
    minVal = a
}
else{
    minVal = b;
}
console.log(minVal);
```

```
let minval = a<b ?  a : b;
console.log(minval);
```

2. Conditional Statements

2.1 Switch Statement

- A Switch statement is a conditional statement like if...else statement used in decision making.

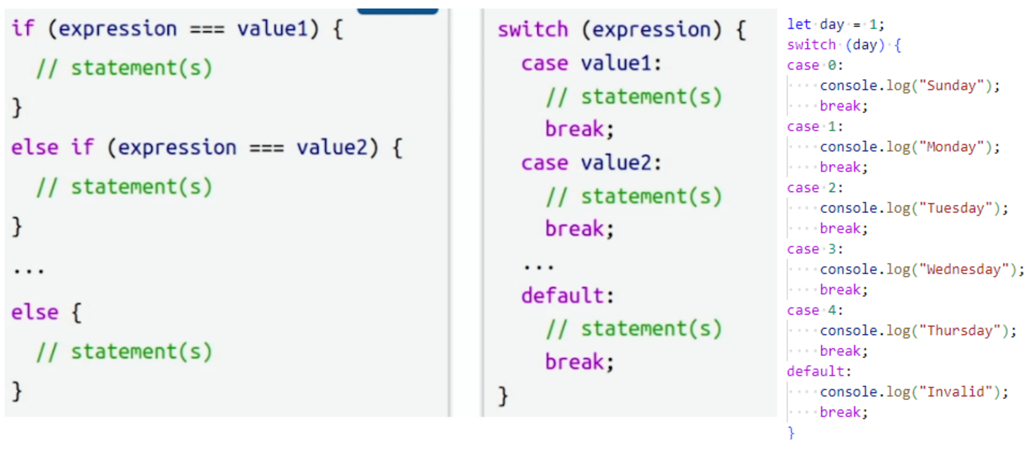
2.1.1 What happens if we forgot a break?

- If there is no break statement, then the execution continues with the next case until the break statement is met.

In [12]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js72.png")
```

Out[12]:



```
if (expression === value1) {
  // statement(s)
}
else if (expression === value2) {
  // statement(s)
}
...
else {
  // statement(s)
}
```

```
switch (expression) {
  case value1:
    // statement(s)
    break;
  case value2:
    // statement(s)
    break;
  ...
  default:
    // statement(s)
    break;
}
```

```
let day = 1;
switch (day) {
case 0:
    console.log("Sunday");
    break;
case 1:
    console.log("Monday");
    break;
case 2:
    console.log("Tuesday");
    break;
case 3:
    console.log("Wednesday");
    break;
case 4:
    console.log("Thursday");
    break;
default:
    console.log("Invalid");
    break;
}
```

3. Defining Functions

- There are multiple ways to define a function.
  - Function Declaration
  - Function Expression
  - Arrow Functions
  - Function Constructor, etc.

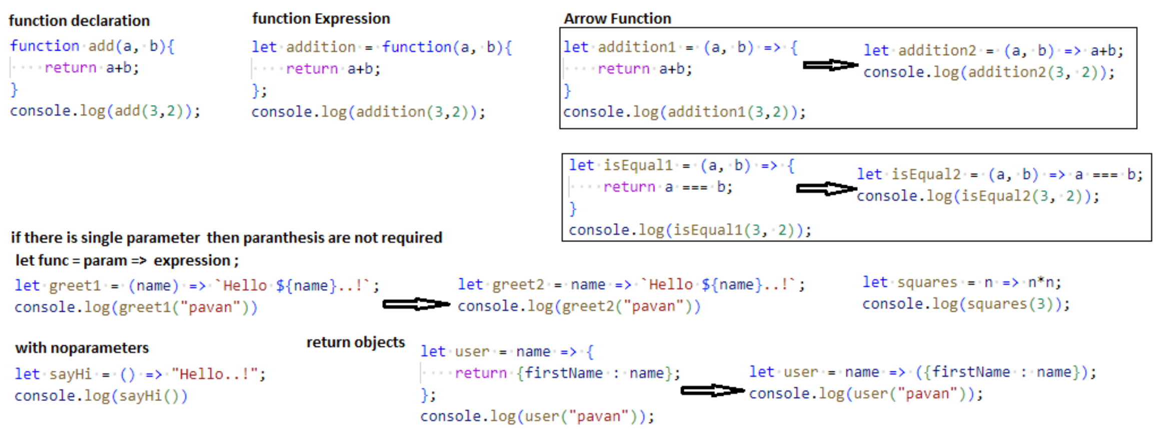
3.1 Arrow Functions

- An Arrow function is a simple and concise syntax for defining functions.
- It is an alternative to a function expression.
- syntax :
  let sum = (param1, param2, …) => { // statement(s) }; sum();

In [1]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js73.png")
```

Out[1]:



function declaration

```
function add(a, b){
    return a+b;
}
console.log(add(3,2));
```

function Expression

```
let addition = function(a, b){
    return a+b;
};
console.log(addition(3,2));
```

Arrow Function

```
let addition1 = (a, b) => {
    return a+b;
}
console.log(addition1(3,2));
```

$\Rightarrow$

```
let addition2 = (a, b) => a+b;
console.log(addition2(3, 2));
```

```
let isEqual1 = (a, b) => {
    return a === b;
}
console.log(isEqual1(3, 2));
```

$\Rightarrow$

```
let isEqual2 = (a, b) => a === b;
console.log(isEqual2(3, 2));
```

if there is single parameter then paranthesis are not required
let func = param => expression ;

```
let greet1 = (name) => `Hello ${name}..!`;
console.log(greet1("pavan"))
```

$\Rightarrow$

```
let greet2 = name => `Hello ${name}..!`;
console.log(greet2("pavan"))
```

```
let squares = n => n*n;
console.log(squares(3));
```

with noparameters

```
let sayHi = () => "Hello..!";
console.log(sayHi())
```

return objects

```
let user = name => {
    return {firstName : name};
};
console.log(user("pavan"));
```

$\Rightarrow$

```
let user = name => ({firstName : name});
console.log(user("pavan"));
```

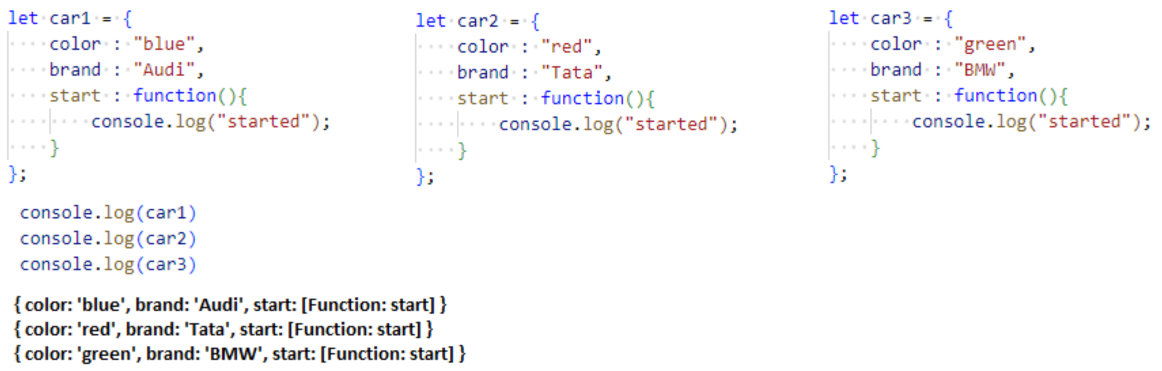
Factory and Constructor Functions

1. object literals

In [3]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js74.png")
```

Out[3]:



```
let car1 = {
    color : "blue",
    brand : "Audi",
    start : function(){
        console.log("started");
    }
};
```

```
let car2 = {
    color : "red",
    brand : "Tata",
    start : function(){
        console.log("started");
    }
};
```

```
let car3 = {
    color : "green",
    brand : "BMW",
    start : function(){
        console.log("started");
    }
};
```

```
console.log(car1)
console.log(car2)
console.log(car3)
```

```
{ color: 'blue', brand: 'Audi', start: [Function: start] }
{ color: 'red', brand: 'Tata', start: [Function: start] }
{ color: 'green', brand: 'BMW', start: [Function: start] }
```

2. Factory Function

- A Factory function is any function that returns a new object for every function call.
- The Function name should follow the camelCase naming convention.

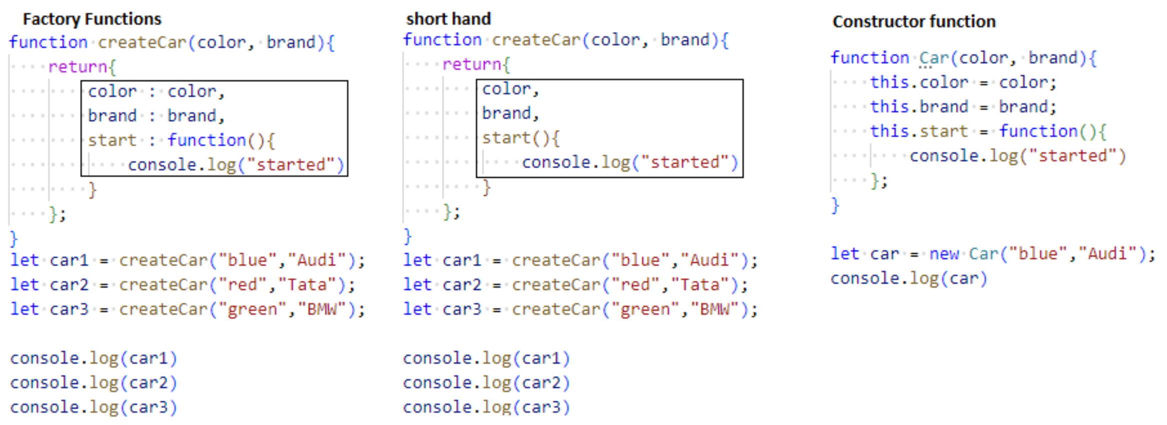
3. Constructor Function

- A regular function that returns a new object on calling with the new operator. The created new object is called an Instance.
- The Function name should follow the PascalCase naming convention.

In [6]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js75.png")
```

Out[6]:



Factory Functions

```
function createCar(color, brand){
    return{
        color : color,
        brand : brand,
        start : function(){
            console.log("started")
        }
    };
}
let car1 = createCar("blue","Audi");
let car2 = createCar("red","Tata");
let car3 = createCar("green","BMW");

console.log(car1)
console.log(car2)
console.log(car3)
```

short hand

```
function createCar(color, brand){
    return{
        color,
        brand,
        start(){
            console.log("started")
        }
    };
}
let car1 = createCar("blue","Audi");
let car2 = createCar("red","Tata");
let car3 = createCar("green","BMW");

console.log(car1)
console.log(car2)
console.log(car3)
```

Constructor function

```
function Car(color, brand){
    this.color = color;
    this.brand = brand;
    this.start = function(){
        console.log("started")
    };
}

let car = new Car("blue","Audi");
console.log(car)
```

Here,

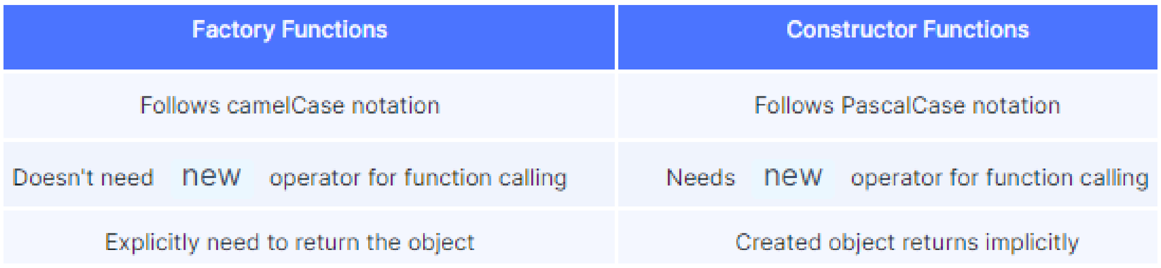
- car1 is instance
- car1.start() is instance method
- car1.color, car1.brand are instance properties

Factory vs Constructor Functions

In [7]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js76.png")
```

Out[7]:



Factory Functions	Constructor Functions
Follows camelCase notation	Follows PascalCase notation
Doesn't need `new` operator for function calling	Needs `new` operator for function calling
Explicitly need to return the object	Created object returns implicitly

3. JS Functions

- Similar to Objects, Functions also have properties and methods.

3.1 Default Properties

- name : This property returns the name of the function.
- length : This property returns the number of parameters passed to the function.
- constructor : Every object in JavaScript has a constructor property.The constructor property refers to the constructor function that is used to create the object.
- prototype, etc.

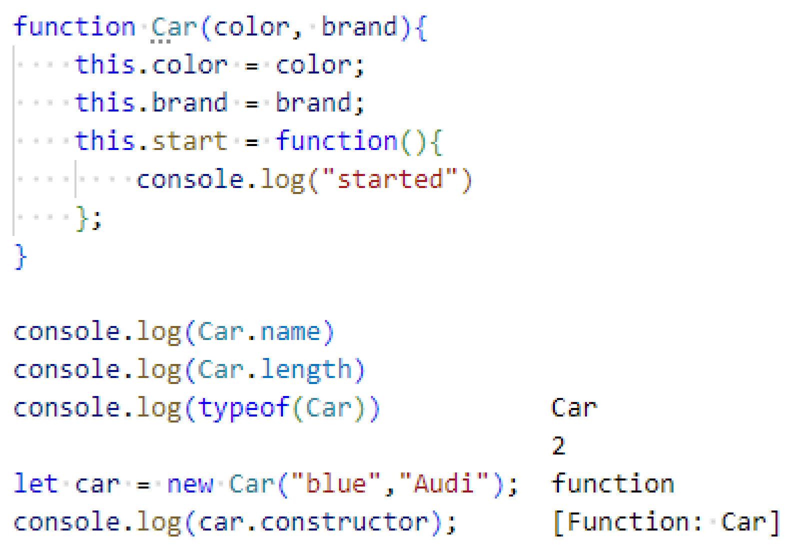
3.2 Default Methods

- apply()
- bind()
- call()
- toString(), etc.

In [9]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js77.png")
```

Out[9]:



```
function Car(color, brand){
    this.color = color;
    this.brand = brand;
    this.start = function(){
        console.log("started")
    };
}

console.log(Car.name)
console.log(Car.length)
console.log(typeof(Car))          Car
                                  2
let car = new Car("blue","Audi"); function
console.log(car.constructor);     [Function: Car]
```

5. Built-in Constructor Function

- These are the Constructor functions provided by JavaScript.
  - function Date()
  - function Error()
  - function Promise()
  - function Object()
  - function String()
  - function Number(), etc.
- In JavaScript, date and time are represented by the Date object. The Date object provides the date and time information and also provides various methods.

5.1 Creating Date Objects

- There are four ways to create a date object.
  - new Date()
  - new Date(milliseconds)
  - new Date(datestring)
  - new Date(year, month, day, hours, minutes, seconds, milliseconds)

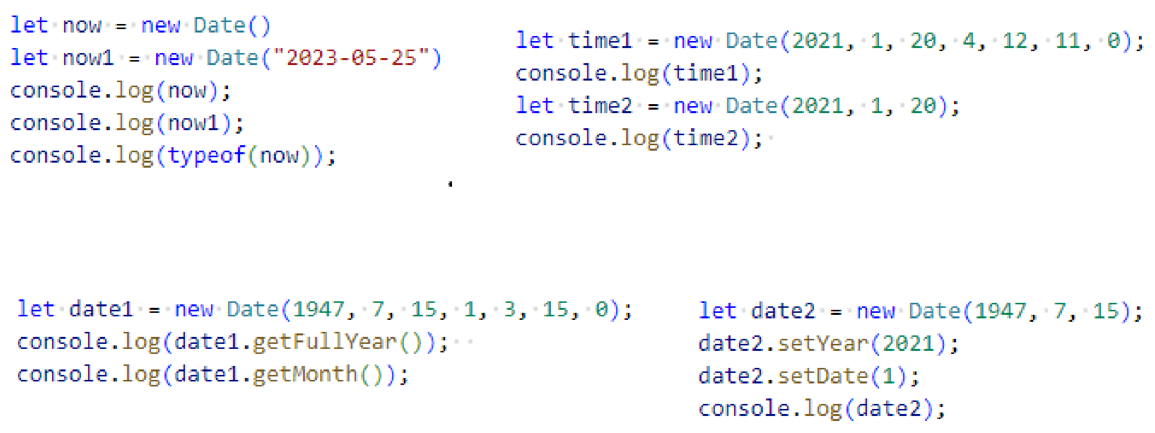
5.3 Instance Methods

- There are methods to access and set values like a year, month, etc. in the Date Object.
- now() Returns the numeric value corresponding to the current time (the number of milliseconds passed since January 1, 1970, 00:00:00 UTC)
- getFullYear() Gets the year according to local time
- getMonth() Gets the month, from 0 to 11 according to local time
- getDate() Gets the day of the month (1–31) according to local time
- getDay() Gets the day of the week (0-6) according to local time
- getHours() Gets the hour from 0 to 23 according to local time
- getMinutes Gets the minute from 0 to 59 according to local time
- getUTCDate() Gets the day of the month (1–31) according to universal time
- setFullYear() Sets the full year according to local time
- setMonth() Sets the month according to local time
- setDate() Sets the day of the month according to local time
- setUTCDate() Sets the day of the month according to universal time

In [10]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js78.png")
```

Out[10]:



```
let now = new Date()
let now1 = new Date("2023-05-25")
console.log(now);
console.log(now1);
console.log(typeof(now));
```

```
let time1 = new Date(2021, 1, 20, 4, 12, 11, 0);
console.log(time1);
let time2 = new Date(2021, 1, 20);
console.log(time2);
```

```
let date1 = new Date(1947, 7, 15, 1, 3, 15, 0);
console.log(date1.getFullYear());
console.log(date1.getMonth());
```

```
let date2 = new Date(1947, 7, 15);
date2.setYear(2021);
date2.setDate(1);
console.log(date2);
```

More Modern JS Concepts

1. this

- The this is determined in three ways.

In Object methods : it refers to the object that is executing the current function.

- In the below example, this refers to the car object as it's executing the method start.

In Regular functions : it refers to the window object.

- In the above example, this refers to the window object.

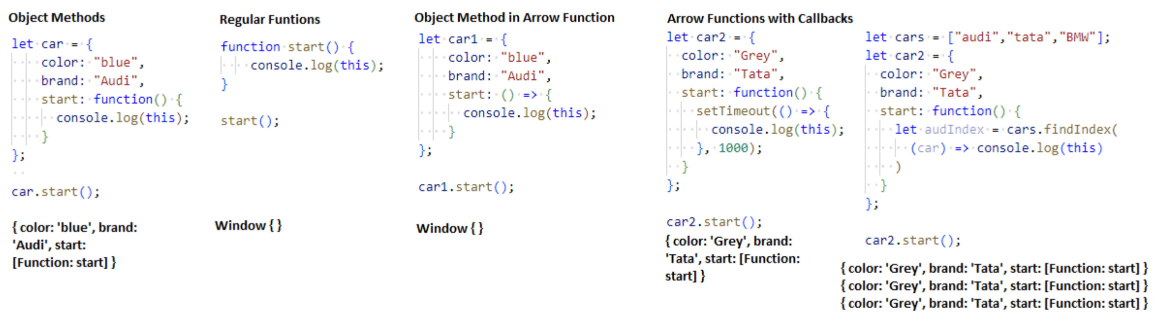
In Arrow functions : it refers to the context in which the code is defined.

- In Arrow functions, this depends on two aspects:
  - When the code is defined
  - Context
- Arrow function inherits this from the context in which the code is defined.

In [2]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js79.png")
```

Out[2]:



Object Methods

```
let car = {
    color: "blue",
    brand: "Audi",
    start: function() {
      console.log(this);
    }
};

car.start();
```

{ color: 'blue', brand: 'Audi', start: [Function: start] }

Regular Funtions

```
function start() {
    console.log(this);
}

start();
```

Window { }

Object Method in Arrow Function

```
let car1 = {
    color: "blue",
    brand: "Audi",
    start: () => {
      console.log(this);
    }
};

car1.start();
```

Window { }

Arrow Functions with Callbacks

```
let car2 = {
  color: "Grey",
  brand: "Tata",
  start: function() {
    setTimeout(() => {
      console.log(this);
    }, 1000);
  }
};

car2.start();
```

{ color: 'Grey', brand: 'Tata', start: [Function: start] }

```
let cars = ["audi","tata","BMW"];
let car2 = {
  color: "Grey",
  brand: "Tata",
  start: function() {
    let audIndex = cars.findIndex(
      (car) => console.log(this)
    )
  }
};

car2.start();
```

{ color: 'Grey', brand: 'Tata', start: [Function: start] }
{ color: 'Grey', brand: 'Tata', start: [Function: start] }
{ color: 'Grey', brand: 'Tata', start: [Function: start] }

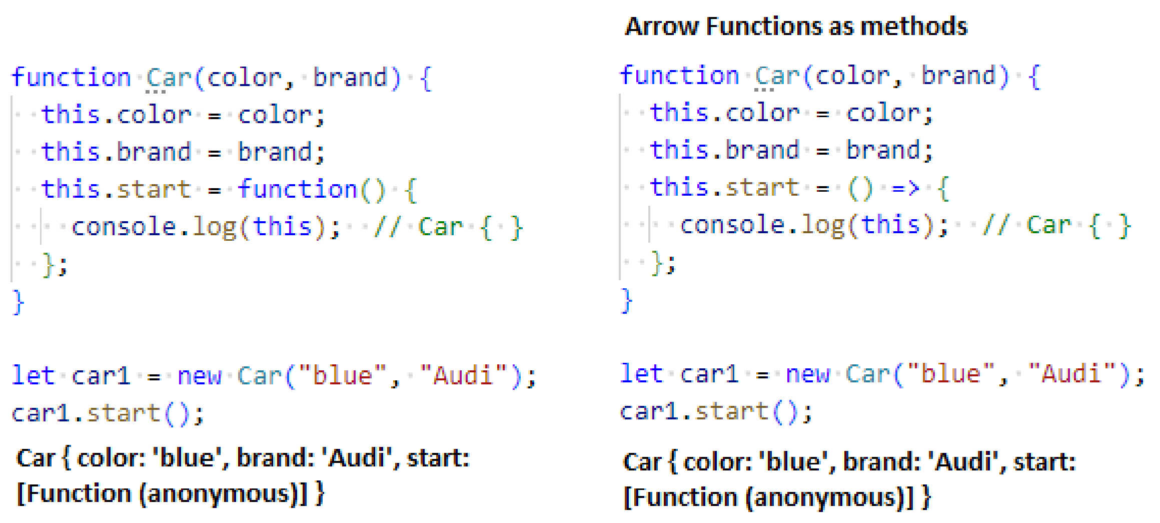
1.4 this in Constructor Functions : In Constructor Function, this refers to the instance object.

- In the below example, this refers to the object car1.
- arrow function inherits this from the context in which the code is defined.

In [3]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js80.png")
```

Out[3]:



```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = function() {
    console.log(this);  // Car { }
  };
}

let car1 = new Car("blue", "Audi");
car1.start();
```

Car { color: 'blue', brand: 'Audi', start: [Function (anonymous)] }

Arrow Functions as methods

```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = () => {
    console.log(this);  // Car { }
  };
}

let car1 = new Car("blue", "Audi");
car1.start();
```

Car { color: 'blue', brand: 'Audi', start: [Function (anonymous)] }

2. Immutable and Mutable Values

2.1 Immutable

- All the primitive type values are immutable.
    - Number
    - String
    - Boolean
    - Symbol
    - Undefined, etc.

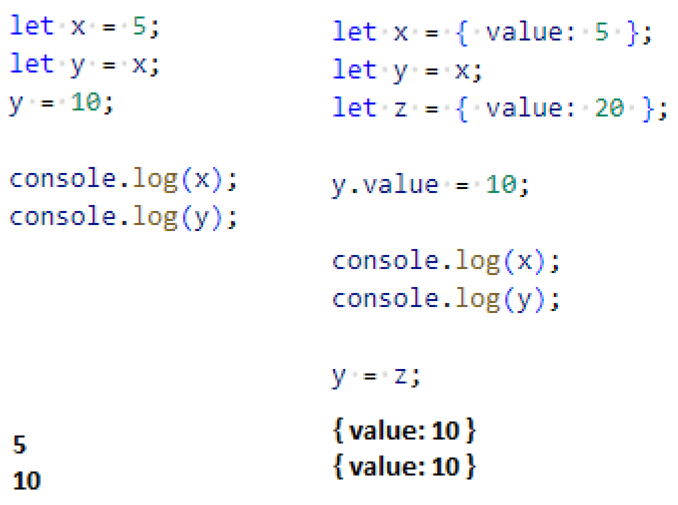
2.2 Mutable

- All the Objects are mutable.
    - Object
    - Array
    - Function

In [4]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js81.png")
```

Out[4]:



```
let x = 5;
let y = x;
y = 10;

console.log(x);
console.log(y);

5
10
```

```
let x = { value: 5 };
let y = x;
let z = { value: 20 };

y.value = 10;

console.log(x);
console.log(y);

y = z;

{ value: 10 }
{ value: 10 }
```

3. Declaring Variables

- In JavaScript, a variable can be declared in 3 ways.
  - Using let
  - Using const
  - Using var

3.1 let

- While declaring variables using let,
  - Initialization is not mandatory
  - Variables can be reassigned

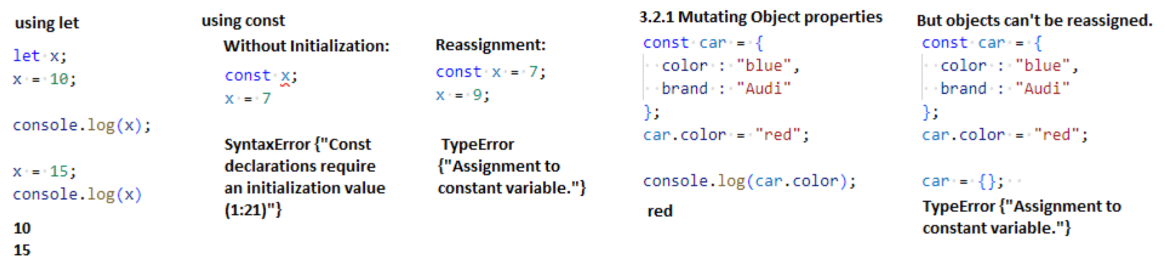
3.2 const

- While declaring variables using const,
  - Initialization is mandatory
  - Once a value is initialized, then it can't be reassigned

In [5]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js82.png")
```

Out[5]:



using let

```
let x;
x = 10;

console.log(x);

x = 15;
console.log(x)
```

```
10
15
```

using const

Without Initialization:

```
const x;
x = 7
```

SyntaxError {"Const declarations require an initialization value (1:21)"}

Reassignment:

```
const x = 7;
x = 9;
```

TypeError {"Assignment to constant variable."}

3.2.1 Mutating Object properties

```
const car = {
  color : "blue",
  brand : "Audi"
};
car.color = "red";

console.log(car.color);
```

```
red
```

But objects can't be reassigned.

```
const car = {
  color : "blue",
  brand : "Audi"
};
car.color = "red";

car = {};
```

TypeError {"Assignment to constant variable."}

Prototypal Inheritance

1. Built-in Constructor Functions

- These are the built-in constructor functions provided by JavaScript.
  - function Array()
  - function Function()
  - function Promise()
  - function Object()
  - function String()
  - function Number(), etc.

2. Built-in Array Constructor Function

2.1 Default Properties and Methods

- Properties:
  - constructor
  - length
  - prototype, etc.
- Methods:
  - push()
  - pop()
  - splice()
  - shift(), etc.

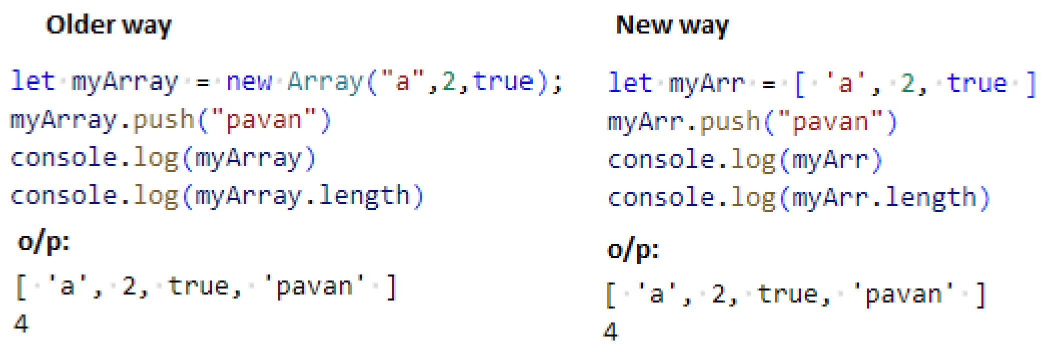
2.2 Creating an Array with the new Operator (Older way of writing)

- Syntax: let myArray = new Array(item1, item2, ...);

In [6]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js83.png")
```

Out[6]:



Older way

```
let myArray = new Array("a",2,true);
myArray.push("pavan")
console.log(myArray)
console.log(myArray.length)
```

o/p:

```
[ 'a', 2, true, 'pavan' ]
4
```

New way

```
let myArr = [ 'a', 2, true ]
myArr.push("pavan")
console.log(myArr)
console.log(myArr.length)
```

o/p:

```
[ 'a', 2, true, 'pavan' ]
4
```

3. Prototype Property

- The Prototype property will be shared across all the instances of their constructor function.

3.1 Accessing the Prototype of a Constructor Function

- console.log(Array.prototype);

3.2 Accessing the shared Prototype of an Instance

- let myArray = new Array("a", 2, true);
- console.log(Object.getPrototypeOf(myArray));

3.3 Prototypal Inheritance

- On calling the new() operator, all the properties and methods defined on the prototype will become accessible to the instance objects. This process is called Prototypal Inheritance.

4. Built-in Function Constructor Function

4.1 Default Properties and Methods

- Properties:
  - name
  - length
  - constructor
  - prototype, etc.
- Methods:
  - apply()
  - bind()
  - call()
  - toString(), etc.

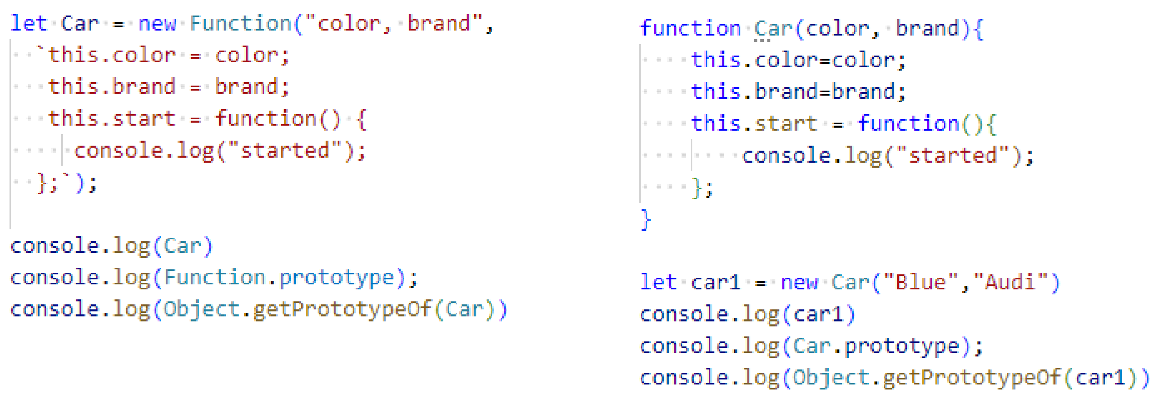
4.2 Creating a Function with the new Operator (Older way of writing)

- Syntax: let myFunction = new Function("param1, param2, ...", function body);

In [7]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js84.png")
```

Out[7]:



```
let Car = new Function("color, brand",
  `this.color = color;
  this.brand = brand;
  this.start = function() {
    console.log("started");
  };`);

console.log(Car)
console.log(Function.prototype);
console.log(Object.getPrototypeOf(Car))
```

```
function Car(color, brand){
    this.color=color;
    this.brand=brand;
    this.start = function(){
        console.log("started");
    };
}

let car1 = new Car("Blue","Audi")
console.log(car1)
console.log(Car.prototype);
console.log(Object.getPrototypeOf(car1))
```

5. Instance Specific and Prototype Properties

5.1 Prototype Properties/ Methods

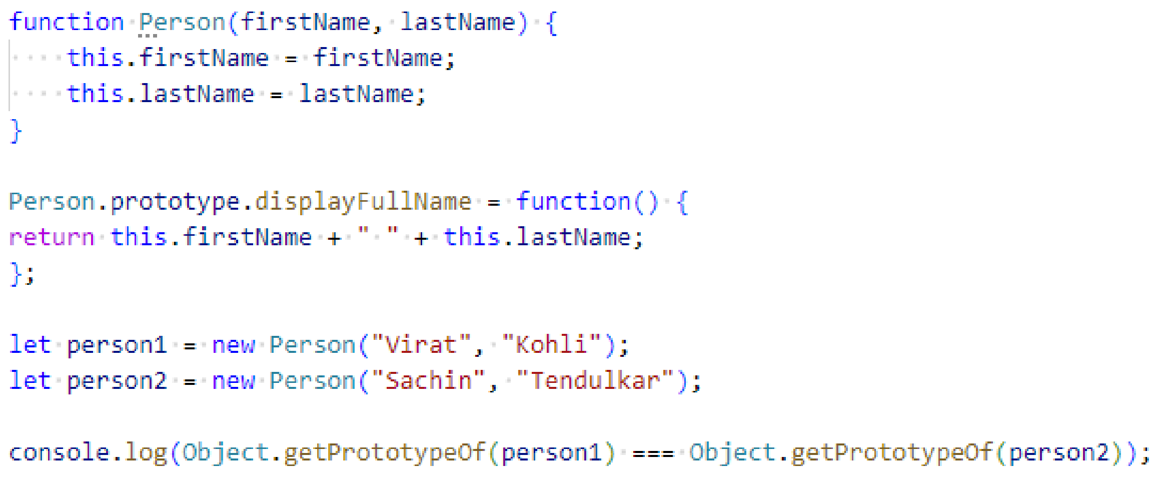
- The Prototype Properties/ Methods are the properties or methods common across the instance objects.
- Examples:
  - calculateAge
  - displayGreetings
  - displayProfileDetails
  - calculateIncome

5.1.1 Adding a Method to the prototype

In [8]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js85.png")
```

Out[8]:



```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

Person.prototype.displayFullName = function() {
return this.firstName + " " + this.lastName;
};

let person1 = new Person("Virat", "Kohli");
let person2 = new Person("Sachin", "Tendulkar");

console.log(Object.getPrototypeOf(person1) === Object.getPrototypeOf(person2));
```

5.2 Instance Specific Properties/ Methods

- The Instance Specific Properties/ Methods are the properties or methods specific to the instance object.
- Examples:
  - gender
  - yearOfBirth
  - friendsList
  - name

JS Classes

1. Class

- The class is a special type of function used for creating multiple objects.

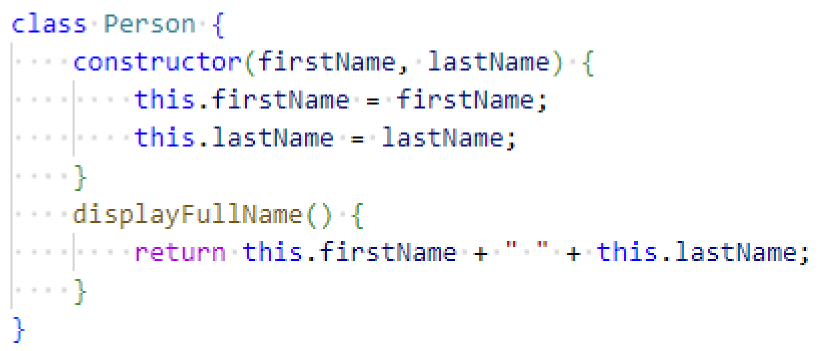
1.1. Constructor Method

- The constructor method is a special method of a class for creating and initializing an object of that class.
- 1.1.1 Creating a Single Object
- 1.1.2 Creating Multiple Objects
- 1.2 Prototype property of a Class

In [11]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js86.png")
```

Out[11]:



```
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    displayFullName() {
        return this.firstName + " " + this.lastName;
    }
}

let person1 = new Person("Virat", "Kohli");
let person2 = new Person("Sachin", "Tendulkar");

console.log(person1)
console.log(person1.displayFullName())
console.log(Object.getPrototypeOf(person1))
console.log(person2)
```

2.Inheritance in JS Classes

- The Inheritance is a mechanism by which a class inherits methods and properties from another class.

2.1 Extends

- The extends keyword is used to inherit the methods and properties of the superclass.

2.2 Super

- Calling super() makes sure that SuperClass constructor() gets called and initializes the instance.
- Here, SubClass inherits methods and properties from a SuperClass.

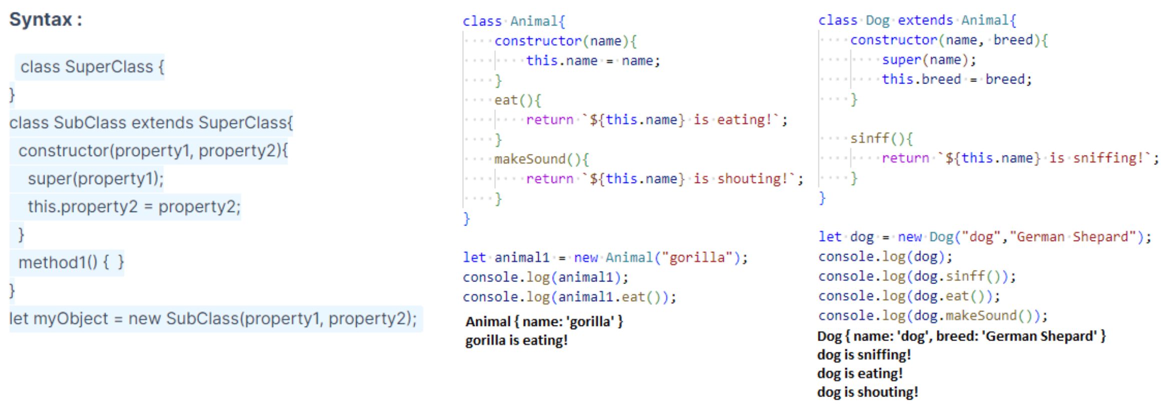
2.3 Method Overriding

- Here the constructor method is overridden. If we write the SuperClass methods in SubClass, it is called method overriding.

In [13]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js87.png")
```

Out[13]:



Syntax :

```
class SuperClass {
}
class SubClass extends SuperClass{
  constructor(property1, property2){
    super(property1);
    this.property2 = property2;
  }
  method1() { }
}
let myObject = new SubClass(property1, property2);
```

```
class Animal{
    constructor(name){
        this.name = name;
    }
    eat(){
        return `${this.name} is eating!`;
    }
    makeSound(){
        return `${this.name} is shouting!`;
    }
}

let animal1 = new Animal("gorilla");
console.log(animal1);
console.log(animal1.eat());
```

```
Animal { name: 'gorilla' }
gorilla is eating!
```

```
class Dog extends Animal{
    constructor(name, breed){
        super(name);
        this.breed = breed;
    }

    sinff(){
        return `${this.name} is sniffing!`;
    }
}

let dog = new Dog("dog","German Shepard");
console.log(dog);
console.log(dog.sinff());
console.log(dog.eat());
console.log(dog.makeSound());
```

```
Dog { name: 'dog', breed: 'German Shepard' }
dog is sniffing!
dog is eating!
dog is shouting!
```

3.this in classes

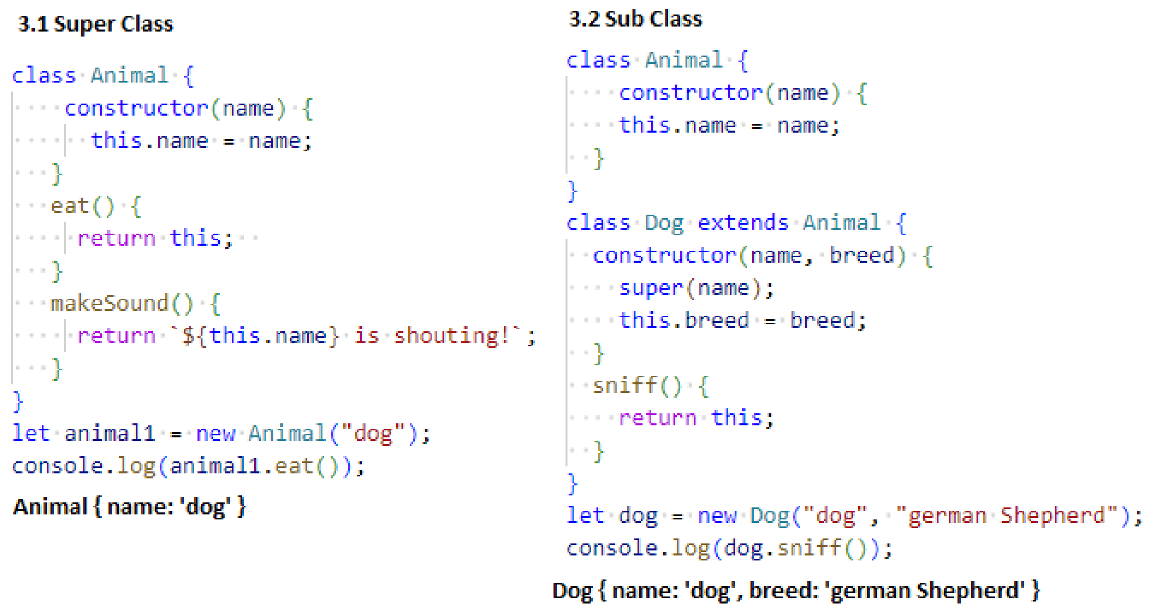
3.1 Super Class

- In class, this refers to the instance object.

In [14]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js88.png")
```

Out[14]:



3.1 Super Class

```
class Animal {
    constructor(name) {
      this.name = name;
    }
    eat() {
      return this;
    }
    makeSound() {
      return `${this.name} is shouting!`;
    }
}
let animal1 = new Animal("dog");
console.log(animal1.eat());
```

```
Animal { name: 'dog' }
```

3.2 Sub Class

```
class Animal {
    constructor(name) {
    this.name = name;
  }
}
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
  sniff() {
    return this;
  }
}
let dog = new Dog("dog", "german Shepherd");
console.log(dog.sniff());
```

```
Dog { name: 'dog', breed: 'german Shepherd' }
```

JS Promises

1. Synchronous Execution

- The code executes line by line. This behavior is called synchronous behavior, in JS alert works synchronously.
- Example :
  - alert("First Line");
  - alert("Second Line");
  - alert("Third Line");

2. Asynchronous Execution

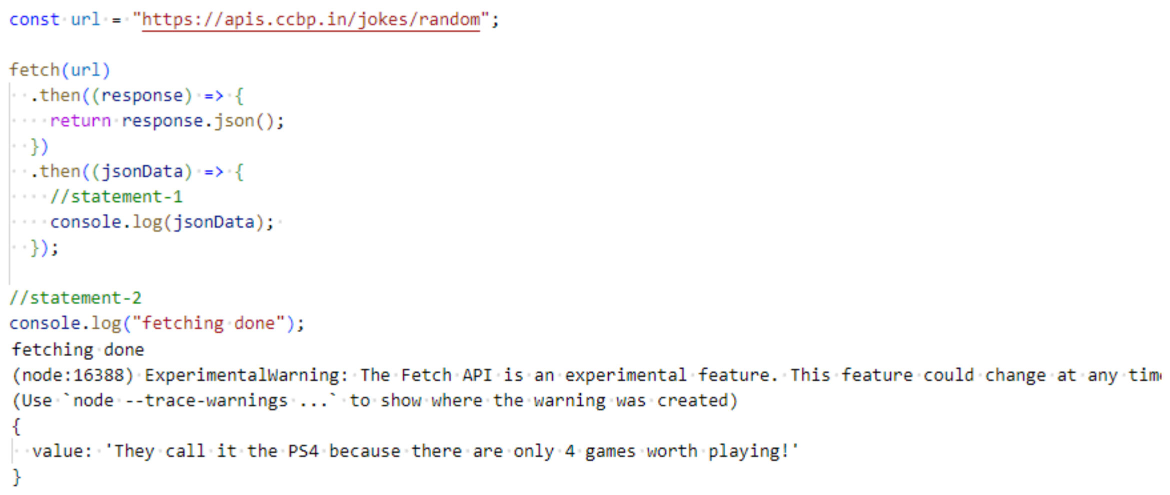
- In the below example, the second statement won't wait until the first statement execution. In JS, fetch() works asynchronously.

Example 1:

In [18]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js89.png")
```

Out[18]:



```
const url = "https://apis.ccbp.in/jokes/random";

fetch(url)
  .then((response) => {
    return response.json();
  })
  .then((jsonData) => {
    //statement-1
    console.log(jsonData);
  });

//statement-2
console.log("fetching done");
fetching done
(node:16388) ExperimentalWarning: The Fetch API is an experimental feature. This feature could change at any tim
(Use `node --trace-warnings ...` to show where the warning was created)
{
  value: 'They call it the PS4 because there are only 4 games worth playing!'
}
```

Synchronous vs Asynchronous

- In Synchronous, the second statement of code executes only after the completion of the first statement
- In Asynchronous, the second statement won't wait until the first statement execution.
  - Asynchronous operations like fetching resource from web
  - reading large files
  - Retrieving the device's current location

3. JS Promises

- Promise is a way to handle Asynchronous operations.
- A promise is an object that represents a result of operation that will be returned at some point in the future.

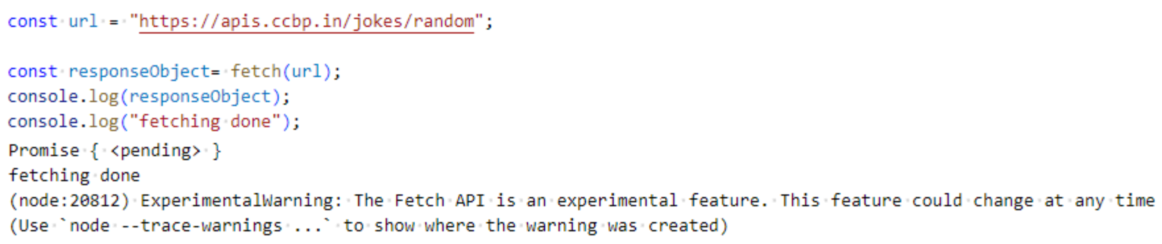
A promise will be in any one of the three states:

- Pending : Neither fulfilled nor rejected
- Fulfilled : Operation completed successfully
- Rejected : Operation failed

In [24]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js90.png")
```

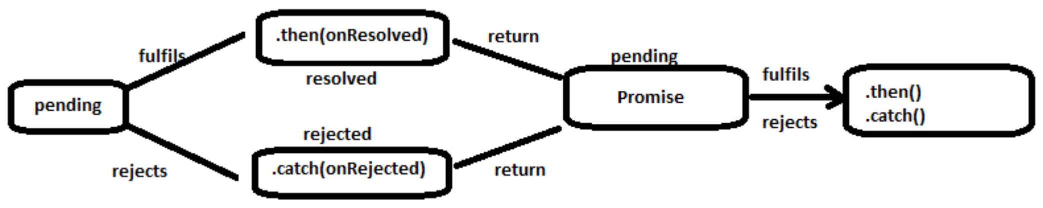
Out[24]:



```
const url = "https://apis.ccbp.in/jokes/random";

const responseObject= fetch(url);
console.log(responseObject);
console.log("fetching done");
```

```
Promise { <pending> }
fetching done
(node:20812) ExperimentalWarning: The Fetch API is an experimental feature. This feature could change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
```



3.1 Resolved State

- When a Promise object is Resolved, the result is a value.

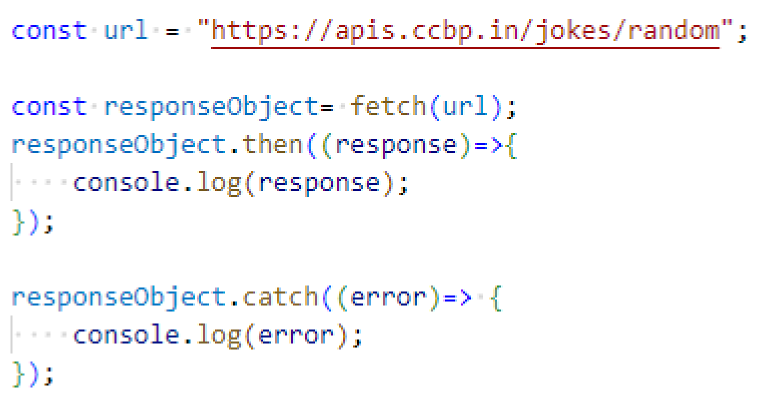
3.2 Rejected State

- Fetching a resource can be failed for various reasons like:
  - URL is spelled incorrectly
  - Server is taking too long to respond
  - Network failure error, etc.

In [21]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js91.png")
```

Out[21]:



```
const url = "https://apis.ccbp.in/jokes/random";

const responseObject= fetch(url);
responseObject.then((response)=>{
    console.log(response);
});

responseObject.catch((error)=> {
    console.log(error);
});
```

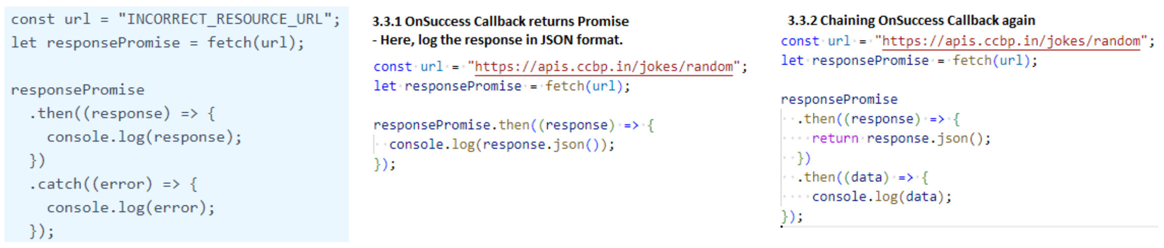
3.3 Promise Chaining

- Combining multiple .then()s or .catch()s to a single promise is called promise chaining.

In [22]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js92.png")
```

Out[22]:



```
const url = "INCORRECT_RESOURCE_URL";
let responsePromise = fetch(url);

responsePromise
  .then((response) => {
    console.log(response);
  })
  .catch((error) => {
    console.log(error);
  });
```

3.3.1 OnSuccess Callback returns Promise
- Here, log the response in JSON format.

```
const url = "https://apis.ccbp.in/jokes/random";
let responsePromise = fetch(url);

responsePromise.then((response) => {
  console.log(response.json());
});
```

3.3.2 Chaining OnSuccess Callback again

```
const url = "https://apis.ccbp.in/jokes/random";
let responsePromise = fetch(url);

responsePromise
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
});
```

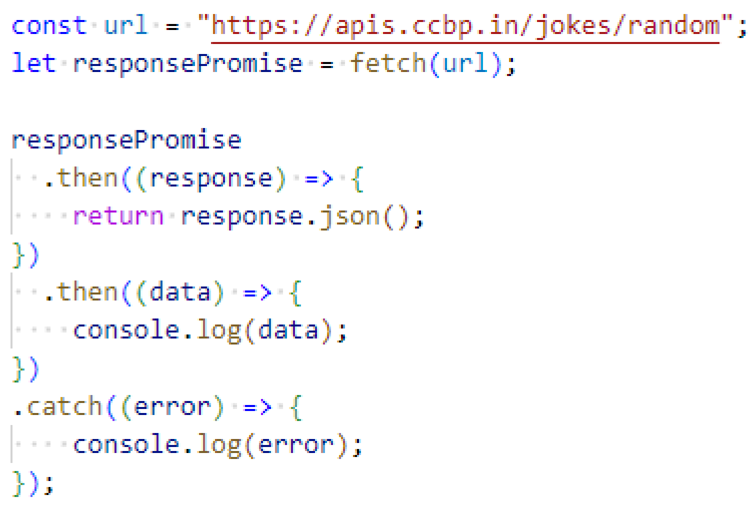
3.4 Fetch with Error Handling

- Check the behavior of code with valid and invalid URLs

In [23]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js93.png")
```

Out[23]:



```
const url = "https://apis.ccbp.in/jokes/random";
let responsePromise = fetch(url);

responsePromise
  .then((response) => {
    return response.json();
})
  .then((data) => {
    console.log(data);
})
.catch((error) => {
    console.log(error);
});
```

JS Promises | Part 2

1. Asynchronous JS code style

- There are two main types of asynchronous code style you'll come across in JavaScript:
  - Callback based : Example : setTimeout(), setInterval()
  - Promise based : Example : fetch()

2. Creating your own Promises

- Promises are the new style of async code that you'll see used in modern JavaScript.
- syntax: const myPromise = new Promise((resolveFunction, rejectFunction) => {

```
    //Async task
```

  });

In the above syntax:

- The Promise constructor takes a function (an executor) that will be executed immediately and passes in two functions: resolve, which must be called when the Promise is resolved (passing a result), and reject when it is rejected (passing an error).
- The executor is to be executed by the constructor, during the process of constructing the new Promise object.
- resolveFunction is called on promise fulfilled.
- rejectFunction is called on promise rejection.

2.1 Accessing Arguments from Resolve

- When resolve() is excuted, callback inside then() will be executed.

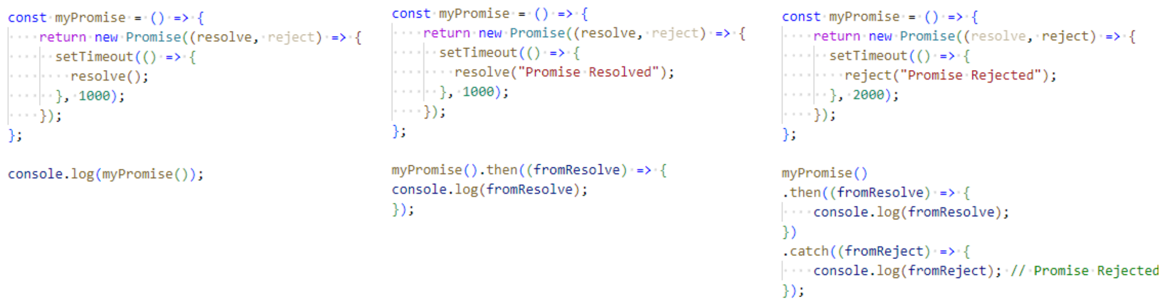
2.2 Accessing Arguments from Reject

- When reject() is excuted, callback inside catch() will be executed.

In [26]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js94.png")
```

Out[26]:



```
const myPromise = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve();
        }, 1000);
    });
};

console.log(myPromise());
```

```
const myPromise = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Promise Resolved");
        }, 1000);
    });
};

myPromise().then((fromResolve) => {
console.log(fromResolve);
});
```

```
const myPromise = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject("Promise Rejected");
        }, 2000);
    });
};

myPromise()
.then((fromResolve) => {
    console.log(fromResolve);
})
.catch((fromReject) => {
    console.log(fromReject); // Promise Rejected
});
```

3. Async/Await

- The Async/Await is a modern way to consume promises.
- The Await ensures processing completes before the next statement executes.

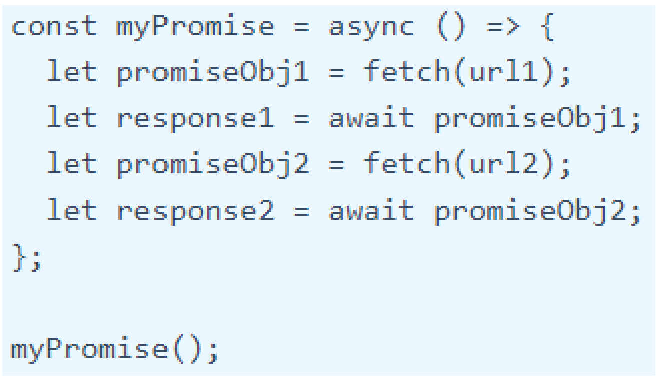
Note

- Use async keyword before the function only if it is performing async operations.
- Should use await inside an async function only.

In [27]:

```python
from IPython.display import Image
Image("E:/code/frontend/img/js95.png")
```

Out[27]:



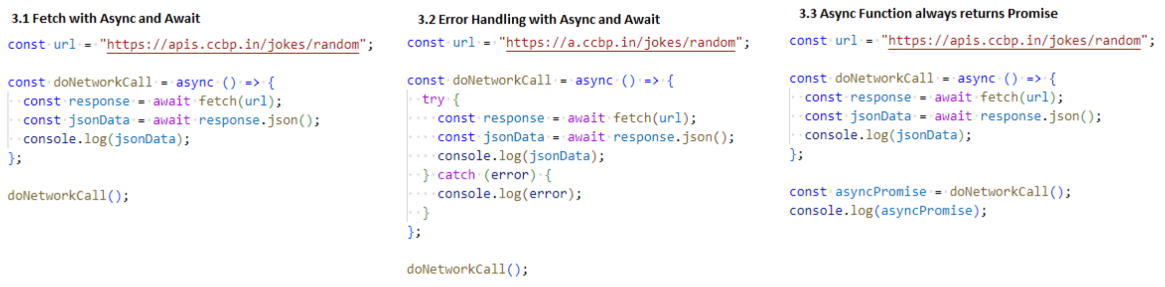
```javascript
const myPromise = async () => {
  let promiseObj1 = fetch(url1);
  let response1 = await promiseObj1;
  let promiseObj2 = fetch(url2);
  let response2 = await promiseObj2;
};

myPromise();
```

In [28]:

```python
from IPython.display import Image
Image("E:/code/frontend/img/js96.png")
```

Out[28]:



3.1 Fetch with Async and Await

```javascript
const url = "https://apis.ccbp.in/jokes/random";

const doNetworkCall = async () => {
  const response = await fetch(url);
  const jsonData = await response.json();
  console.log(jsonData);
};

doNetworkCall();
```

3.2 Error Handling with Async and Await

```javascript
const url = "https://a.ccbp.in/jokes/random";

const doNetworkCall = async () => {
  try {
    const response = await fetch(url);
    const jsonData = await response.json();
    console.log(jsonData);
  } catch (error) {
    console.log(error);
  }
};

doNetworkCall();
```

3.3 Async Function always returns Promise

```javascript
const url = "https://apis.ccbp.in/jokes/random";

const doNetworkCall = async () => {
  const response = await fetch(url);
  const jsonData = await response.json();
  console.log(jsonData);
};

const asyncPromise = doNetworkCall();
console.log(asyncPromise);
```

4. String Manipulations

- There are methods and properties available to all strings in JavaScript.
- toUpperCase(), toLowerCase() : Converts from one case to another
- includes(), startsWith(), endsWith() : Checks a part of the string
- split() : Splits a string
- toString() : Converts number to a string
- trim(), replace() : Updates a string
- concat(), slice(), substring() : Combines & slices strings
- indexOf() : Finds an index

- const greeting = " Hello world! ";
- console.log(greeting);
- console.log(greeting.trim());

More JS Concepts

1. Scoping

- The Scope is the region of the code where a certain variable can be accessed.
- In JavaScript there are two types of scope:
  - Block scope
  - Global scope

1.1 Block Scope

- If a variable is declared with const or let within a curly brace ({}), then it is said to be defined in the Block Scope.
  - if..else
  - function (){}
  - switch
  - for..of, etc.

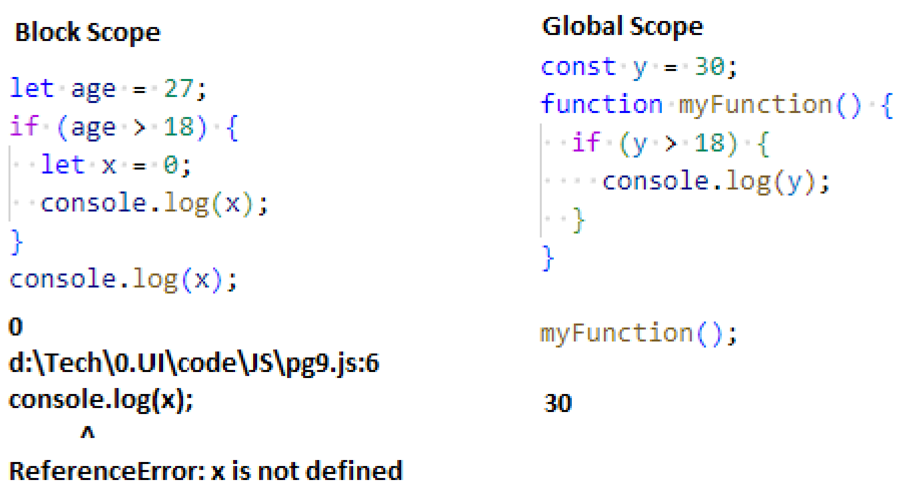
1.2 Global Scope

- If a variable is declared outside all functions and curly braces ({}), then it is said to be defined in the Global Scope.
- When a variable declared with let or const is accessed, Javascript searches for the variable in the block scopes first followed by global scopes.

In [1]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js97.png")
```

Out[1]:



Block Scope

```
let age = 27;
if (age > 18) {
  let x = 0;
  console.log(x);
}
console.log(x);
```

```
0
d:\Tech\0.UI\code\JS\pg9.js:6
console.log(x);
        ^
ReferenceError: x is not defined
```

Global Scope

```
const y = 30;
function myFunction() {
  if (y > 18) {
    console.log(y);
  }
}

myFunction();
```

```
30
```

2. Hoisting

2.1 Function Declarations

- Hoisting is a JavaScript mechanism where function declarations are moved to the top of their scope before code execution.

2.2 Function Expressions

- Function expressions in JavaScript are not hoisted.

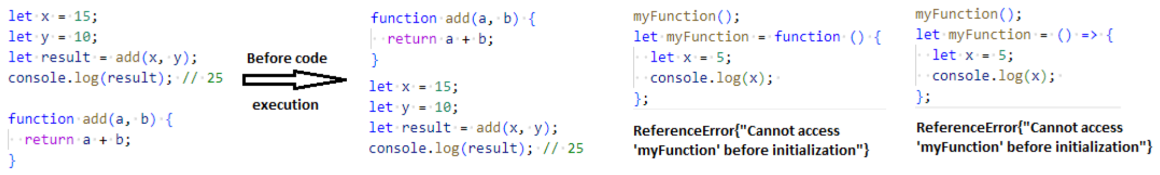
2.3 Arrow Functions

- Arrow Functions in JavaScript are not hoisted.

In [2]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js98.png")
```

Out[2]:



```
let x = 15;
let y = 10;
let result = add(x, y);
console.log(result); // 25

function add(a, b) {
  return a + b;
}
```

```
function add(a, b) {
  return a + b;
}
let x = 15;
let y = 10;
let result = add(x, y);
console.log(result); // 25
```

```
myFunction();
let myFunction = function () {
  let x = 5;
  console.log(x);
};
```

ReferenceError{"Cannot access 'myFunction' before initialization"}

```
myFunction();
let myFunction = () => {
  let x = 5;
  console.log(x);
};
```

ReferenceError{"Cannot access 'myFunction' before initialization"}

3. More Array Methods

- The map(), forEach(), filter() and reverse() are some of the most commonly used array methods in JavaScript.

3.1 Map()

- The map() method creates a new array with the results of calling a function for every array element.
- The map() method calls the provided function once for each element in an array, in order. #### Syntax : array.map(callback(currentValue, index, arr))
- Here the callback is a function that is called for every element of array.
- currentValue is the value of the current element and index is the array index of the current element. Here index and arr are optional arguments.

3.2 forEach()

- The forEach() method executes a provided function once for each array element. It always returns undefined. #### Syntax : array.forEach(callback(currentValue, index, arr))
- Here index and arr are optional arguments.

3.3 filter()

- The filter() method creates a new array filled with all elements that pass the test (provided as a function).
- A new array with the elements that pass the test will be returned. If no elements pass the test, an empty array will be returned. #### Syntax : array.filter(function(currentValue, index, arr))
- Here index and arr are optional arguments.

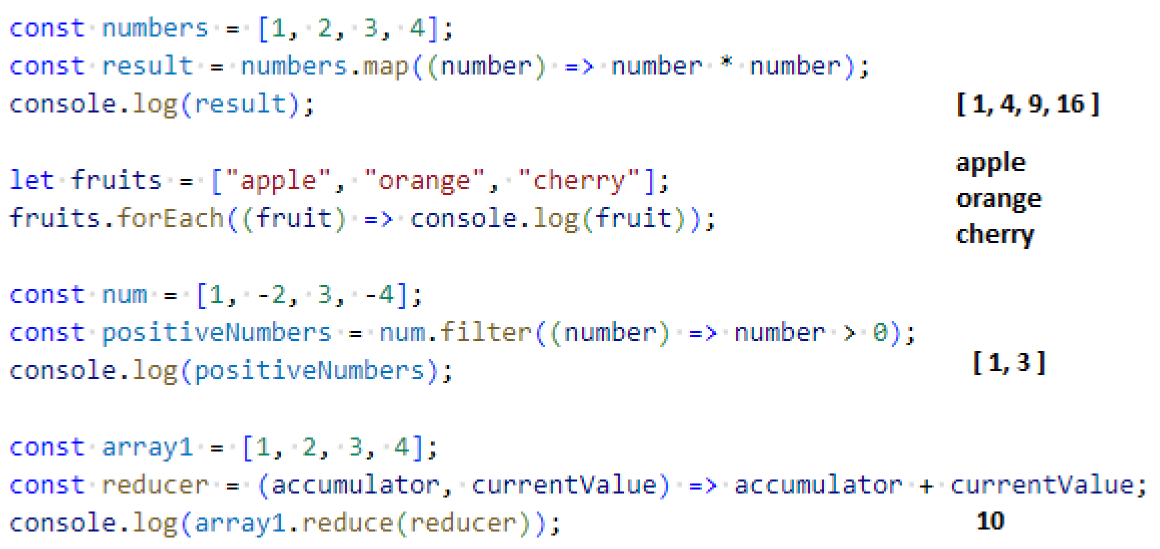
3.4 reduce()

- The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in single output value. #### Syntax : array.reduce(function(accumulator, currentValue, index, arr), initialValue)
- Here accumulator is the initialValue or the previously returned value of the function and currentValue is the value of the current element, index and arr are optional arguments.

In [3]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js99.png")
```

Out[3]:



```
const numbers = [1, 2, 3, 4];
const result = numbers.map((number) => number * number);
console.log(result);                                        [ 1, 4, 9, 16 ]

let fruits = ["apple", "orange", "cherry"];                 apple
fruits.forEach((fruit) => console.log(fruit));              orange
                                                            cherry

const num = [1, -2, 3, -4];
const positiveNumbers = num.filter((number) => number > 0);
console.log(positiveNumbers);                                [ 1, 3 ]

const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;
console.log(array1.reduce(reducer));                         10
```

3.5 every()

- The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

  #### Syntax : array.every(function(currentValue, index, arr))
- Here index and arr are optional arguments.

3.6 some()

- The some() method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

  #### Syntax : array.some(function(currentValue, index, arr))
- Here index and arr are optional arguments.

3.7 reverse()

- The reverse() method reverses the order of the elements in an array.The first array element becomes the last, and the last array element becomes the first. #### Syntax : array.reverse()

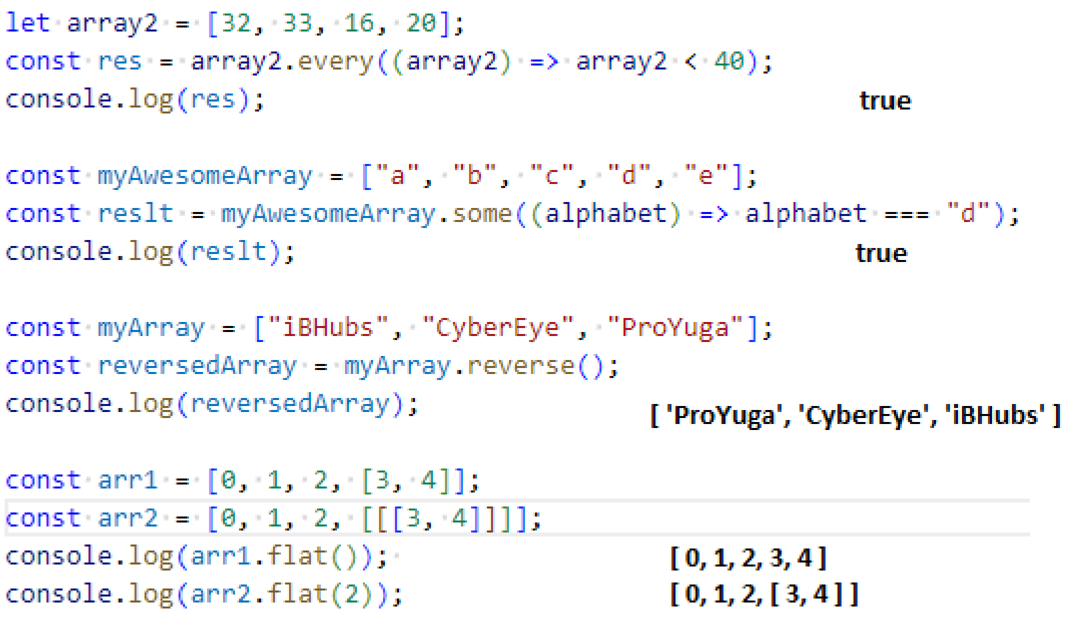
3.8 flat()

- The flat() method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth. #### Syntax : let newArray = arr.flat([depth]);

In [4]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js100.png")
```

Out[4]:



```
let array2 = [32, 33, 16, 20];
const res = array2.every((array2) => array2 < 40);
console.log(res);                                    true

const myAwesomeArray = ["a", "b", "c", "d", "e"];
const reslt = myAwesomeArray.some((alphabet) => alphabet === "d");
console.log(reslt);                                  true

const myArray = ["iBHubs", "CyberEye", "ProYuga"];
const reversedArray = myArray.reverse();
console.log(reversedArray);              [ 'ProYuga', 'CyberEye', 'iBHubs' ]

const arr1 = [0, 1, 2, [3, 4]];
const arr2 = [0, 1, 2, [[[3, 4]]]];
console.log(arr1.flat());                  [ 0, 1, 2, 3, 4 ]
console.log(arr2.flat(2));                 [ 0, 1, 2, [ 3, 4 ] ]
```

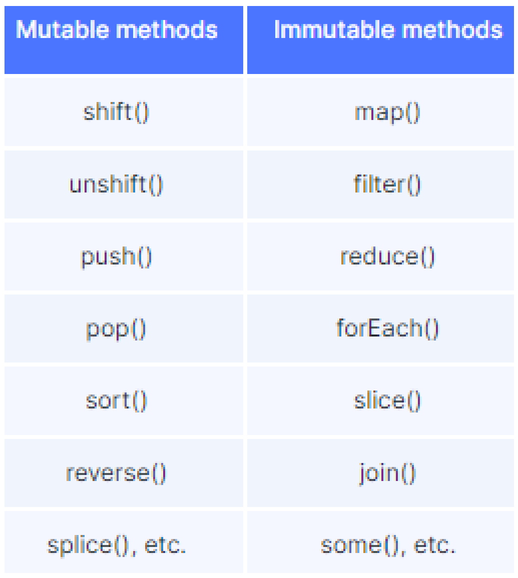
4. Mutable & Immutable methods

- Mutable methods will change the original array and Immutable methods won't change the original array.

In [5]:

```
from IPython.display import Image
Image("E:/code/frontend/img/js101.png")
```

Out[5]:



Mutable methods	Immutable methods
shift()	map()
unshift()	filter()
push()	reduce()
pop()	forEach()
sort()	slice()
reverse()	join()
splice(), etc.	some(), etc.

1. Variable and Function names

1.1 Use intention-revealing names

- const firstName = "Rahul";
- const lastName = "Attuluri";
- console.log(firstName);
- console.log(lastName);

1.2 Make your variable names easy to pronounce

- let firstName, lastName;
- let counter;
- const maxCartSize = 100;
- const isFull = cart.size > maxCartSize;

2. Better Functions

2.1 Less arguments are better

function Circle(center, radius) { this.x = center.x; this.y = center.y; this.radius = radius; }

2.2 Use Arrow Functions when they make code cleaner

const count = 0; const incrementCount = (num) => num + 1;

2.3 Use Async await for asynchronous code

const doAllTasks = async () => { await myPromise; await func1(); await func2(); }; doAllTasks();