Introduction To JavaScript

History of JavaScript

- Brendan Eich created JavaScript in 1995.
- Mocha was the first name of JavaScript.
- It was then changed to LiveScript.
- Finally it was changed to JavaScript.

Microsoft and Internet Explorers role in JavaScript

- At the ime of launch of Internet Explorer Microsoft created new language called JScript using some features of JavaScript and some of their own features.
- So there were two languages JavaScript and JScript.
- JScript was working only on Internet Explorer.
- JavaScript was working on other browsers like NetScape Navigator.

Introduction to ECMA

- Since there were two languages and each were not compartible with all browsers, JavaScript was taken to ECMA to standarize.
- After ECMA standarization the name of JavaScript was changed to ECMAScript.
- Since then ECMA started releasing its versions ES1, ES2, ES3, ES4,ES5.
- Lots of features were added in ES5 version of JavaScript and started gaining popularity.
- Then came the biggest update of JavaScript as ES6. And the name ES6 was changed to ES2015 since it was resleased in 2015. This version is also called modern JavaScript.
- Then yearly updates started coming out with ES7 : ES2016, ES8 : ES2017

Important things to know

- JavaScript is a backward compartible language - means all the features from previous releases will also work in the latest current release.
- JavaScript is not a forward compartible language - means new features will not work in very old versions of web browsers.

Executing a JavaScript Code

Method 1:

- Create a HTML file and inside body tag add script tag spicifying the source of the JavaScript file.
- We basically add JavaScript file just above ending the body tag.
- Then view the output in browsers console.(ctrl +Shift+J to open console in chrome)
- If you want to add script tag in the head section then write defer after spicifying the scr file.

First JavaScript Program

- To print anything in the console we use **console.log()** followed by semicolon.
- Anything written inside the parentheses of console.log with double quotes, single quotes or backticks will be printed in the console.
- Code will also run without semicolon but writing with semicolons at the end of statements is a good practice.

Comments in JavaScript

- We write comments in JavaScript using double slash like //
- Shortcut to comment multiple line is **ctrl+/**.
- To uncomment select and again press **ctrl+/**.

Data Types

- Number
- String
- Boolean
- Undefined
- Null
- BigInt
- Symbol
- Object

Undefined

- This varibales does not have any value assigned to them.
- Undefined variables can be created using var and let.
- You cannot created undefined variable with constant.
- You can assign value to the undefined variables later.

Null

- You can initialize the null variable by giving the variable value as null.
- You can define null variable even with constant.
- You can change the value of null variable later.
- The typeof null variable shows it as a object, that is an error/bug in JavaScript. It is not resolved because all the currently running codes of the websites and framworks will have to be changed.

BigInt

- All numbers which are greater then the maximum safe integer accepted by JavaScript can be considered as BigInt.
- You can write bigInt as shown below

  let a = BigInt(12345);

  let b = 234n;
- Even smaller numbers which are writen inside ***BigInt()*** or those ends with ***'n'*** are considered as BigInt numbers.
- You can perform any operations between two or more BigInt numbers.
- You cannot mix BigInt with other data types to perform operations without converting them to the same type.

Variables

- Variables can store some information and we can use and change that information later.
- Variables and defined using var keyword and the names of variables are case sensitive.
- We can also declare variables without writing var keyword.

Use Strict

The statement "use strict" is used in JavaScript to avoid declaring variables without specifying var keyword to avoid mistakes.

Rules for naming a variable

- You cannot start a variable with a number.
- You can only use underscore (_) or dollar ($) symbol.
- You cannot use spaces in variable names.

You can either use snake case (snake_case) writing or camel case writing. Conventionally its recommended to use camelCase.

Let Keyword

- Let keyword is used to declare a variable.

- Always use let keyword to declare variables instead of var.
- Let keyword allows us to declare a variable using one variable name only once. This avoids mistakes and confusion in longer code.
- But using var you can declare a variable using same name multiple times.
- The main difference betwwen let and var is block scope vs functional scope.

Constant

- The value of constant cannot be changed and can be declared using const keyword.
- Constant is also a variable.

Strings in JavaScript

Sitrings are immutable.

String Indexing

Each character of a string in JavaScript has a indexed number associated with it. The indexing strats from number 0.

You can find the length of a string using varibleName.length.
To get the last index of a string you can use variableName.length-1.

String Methods

- All string methods create a new string when performed on an string without automatically modifying the value of original string.
- You can either store the string created by methods in same variable by specifying it or create a new variable to store it.

Commonly used Methods

- **trim()** - Used to remove empty spaces.
- **toUpperCase()** - Change all characters in a string to uppercase.
- **toLowerCase()** - Change all characters in a string to lowercase.
- **slice()** - Used to display only certain part of a string using the start and end value. The string will start from start value to end at endvalue - 1.

String Concatenation

It means adding or joining two or more strings together. We can concatenate strings using plus operator.

Template Strings

If you want to add values from varibales in a string template or long text, you can do it using template strings.

- First store the template or text in a variable using backticks instead of double quotes or single quotes.
- Use ${variable_name} at the places you want to replace or add the values from variables.

Typeof Operator

Typeof operator is used to know the type of data type stored in a variable.

Converting Number to String

Method 1:

Add empty string with the number.
Example
let a = 12;
a = 12 + “”; //Converted to string
let b = 5 + “”; //Converted to string

Method 2:

Using the string method
Example
let a = 76;
a = String(a);

Conveting String to Number

Method 1:

Add plus(+) before the string
Example
let str1 = “44”;
str1 = +str1; //Converted to number
str2 = +”23”; //Converted to number

Method 2:

Using Number method
Example

let a = “57”;
a = Number(a); //Converted to number

==, ===, !=, !== Operators

The == and != operators only checks and compare the values and not the data type of that value in JavaScript.

In JavaScript to check and compare the values as well as data types we have ===, !== operators.

If else condition

The syntax of If else condition in JavaScript.

```
if(condition){
        Statements;
}else{
        Statements;
}
```

Nested if else

Nested if else condition means more if else conditions inside the if statements or esle statements.

```
if(condition){
        Statements;
        if(condition){
                Statements;
        }else{
                Statements;
        }
}else{
        if(condition){
                Statements;
        }else{
                Statements;
        }
}
```

If else if

Syntax

```
if(condition){
        Statements;
}else if(condition){
        Statements;
}else if(condition){
        Statements;
}else (condition){
        Statements;
}
```

- If any if the else if condition is satisfied, the next else if statement or else statement will not be executed.
- The JavaScript engine will directly jump out of the if else conditions and execute other lines on code.

Ternary Operator

In JavaScript you can check if a condition is true using ternary operator. Its like if else condition but it is used only used in case when you have one statement in if condition and one statement in else condition.

Syntax

let voting = age >= 18 ? “Eligible” : “Not Eligible”

In the above example

- It first checks if the condition is true.
- If true it take the first value after **‘?’**.
- If false it take the value after **‘:’**.

OR and AND operators

- **||** is OR operator in JavaScript.
- **&&** is AND operator in JavaScript.

Switch Statement

Syntax

```
switch(key){
        case value :
```

```
        Statements;
        break;
    default:
        break;
}
```

- When any case is satisfied, to stop the further execution of other cases the break statement at the end of each case is mandatory.
- Without break statement all the switch cases of the satisfactory case will be executed will be executed.

Loops

While loop

While loop is used when you dont know how many times the loop will run.

Syntax

```
while(condition){
    Statements;
}
```

For loop

For loop is used when you know when the loop will end.

Syntax

```
for(initialize index; condition; increment index){
    Statements;
}
```

Break and Continue Keywords

- Break keyword when encountered breaks the loop cycle and jumps out of the loop even if the loop condition is true.
- Continue keyword when encountered skips the current loop cycle but executes the next cycles until the loop condition is false.

Do while loop

Do while loop is used when you want to execute the loop atleast once even if the condition is false.

Syntax

```
do{
    Statements;
}while (condition);
```

Arrays

- Arrays are the collection of items.
- Arrays are of reference type. All the reference types are objects therefore array is also object.
- Any data type can be stored in array.
- You can access and change any element of array using its index number.
- Arrays are mutable and all array methods can change the original array.

Syntax
let arr1 = [1, 3.4, "items", 'w', null];

Array methods

- **Array.isarray(name)** - Checks if the object is of type array or not.
- **arrayName.push("element")** - Adds an element to the array at the end.
- **arrayName.pop()** - Removes an element from the end of an array and returns it.
- **arrayName.unshift("element")** - Adds an element to the array at the starting.
- **arrayName.shift()** - Removes an element from the starting of an array an returrns it.

push() and pop() methods are faster then unshift() and shift() method because unshift and shift has to travel to the starting of the array in order to perform their operations.

Clone Array

You can clone array using the slice() method. The digit 0 inside slice method specifies that array will be cloned from starting index to ending index.
Example :
let arr1 = ["item1"];
let arr2 = arr1.slice(0);

Concat Array

Contcat() method is used to join two arrays together. You can also use concat() method to clone array by creating and empty array and adding another array to it using concat() method.
Example :
let arr1 = ["item1"];
let arr2 = [].concat(arr1);

Spread Operator

- Spread operator is used to clone an array in an elegant way. You can spread any iterable data types using spread operator like strings, array, etc.
- Spread operator cannot spread non iterable data types like integers.

Example :

let arr1 = [“item1”];
let arr2 = [...arr1];
let arr3 = [...”abcd”]; // This will create array with each character as saperate array element.

For loop in Array

You can iterate through the array element using for loop. The length method of array helps in accessing the elements.

Constant Array

- You can create or initailize an array using const keyword.
- The array created using const key is comparatively safer then the array created using let or var keywords. So the best way to create an array is using const.
- You can perform all the array operations using array methods in constant array, but you cannot initailize the constant array again to some other elements.
- We can perform array operations using array methods because using these operations we are not changing the array address stored in the stack but we are changing the array refering to same address located in heap memory.

Example:
const arr1 = [“item1”];
arr1.push(“item2”); // This is valid
arr1 = [“item2]; // This is invalid and will throw an error.

While loop in Array

You can also iterate array elements using while loop. The length method of array helps in accessing the elements.

For of Loop

For of loop is used to iterate elements from an array. This loop directly returns the element of each index from starting to the end of an array.

For of loop is used most of the times and the traditional for loop.
Syntax:
const arr1 = [“item1”];
for (let item of arr1) {
    console.log(item);
}

For in Loop

For in loop is used to iterate element indexes from an array. This loop returns the index numbers of array elements from starting to the end.

Syntax:
const arr1 = [“item1”];
for (let index in arr1) {
    console.log(arr1[index]);
}

Array Destructuring

You can use array destructuring for various reasons.

- You can store array elements in variables using array destructuring. The element will be stored in variables from starting till the last variable. The rest of the array element will not be stored.
  Example:
  const arr1 = [“item1”, “item2”, “item3”, “item4”];
  let [var1, var2] = arr1;

- You can also skip elements while storing in variables.
  Example:
  const arr1 = [“item1”, “item2”, “item3”, “item4”];
  let [var1, , var3] = arr1; // Skipped element at index 1(item2)

- You can store some elements in variables and store rest of the further elements in another array.
  Example:
  const arr1 = [“item1”, “item2”, “item3”, “item4”];
  let [var1, var2, …newArr] = arr1; // Last two element will be stored in newArr array

- If your array size is short and you specified more variables, then the extra variables will be declared as undefined.
  Example:
  const arr1 = [“item1”, “item2”];
  let [var1, var2, var3] = arr1; // var3 will be undefined here

- Variables declared using array destructuring can also be used as normal variables.
  Example:
  const arr1 = [“item1”, “item2”, “item3”, “item4”];
  let [var1, var2] = arr1;
  console.log(var2);
  var2 = “This is changed value”; // Value of var2 here is changed

console.log(var2);

Primitive vs Reference data types

Primitive

- Primitive data types are stored in a stack memory. Each variable in stored separately stacking upon each other.
- If any variables data is chaged it will change the data of only that variable and not of any variable pointing to that particular variable.

Reference

- Reference data types are stored in a heap memory. A reference pointer pointing to the address of each reference data type is stored in stack.
- If we initialize two reference data types as equal to each other, then the reference pointer of both data types will be storing the same reference address pointing to the same data.
- So if we change data of any one reference data type, then the data of others will also change which are initialized as equal to that data type.

Objects in JavaScript

- Objects are used to store real world data since arrays are not sufficient.
- Objects are stored in key value pairs and objects don't have any index number associated to them.
- Objects are of reference type data types like arrays. The objects data is stored same as its being stored in case of arrays.
- You cannot have one key twice in an object. In case of same keys multiple times he value of last encountered key will be assigned to it.

Syntax
const objName { key: "value"}

Example 1:
const person { name: "John", age: 30, hobby: ["hobby1", "hobby2"] }

Example 2:
const person { "name": "John", "age": 30, "hobby": ["hobby1", "hobby2"] }
console.log(person.name);
console.log(person["name"]);

- The key of the object is of type string. So you can also create objects by using double quotes for the key.
- We also call the key of the object as the property of the object.
- We can access the key of the object using dot notation as well as square brackets as the key is of type string. While using square brackets specify the key in double quotes.

Adding Key Value pair to an object.

Example:
const person { name: "John", age: 30, hobby: ["hobby1", "hobby2"] }
person.gender = "male";

Dot notation VS Bracket notation

Bracket notation is helpful when the name of the key has space in between. The dot notation cannot access key names with spaces between them.
Example:
const person { name: "John", age: 30, "person hobby": ["hobby1", "hobby2"] }
console.log(person["person hobby"]);

The other usecase of bracket notation is when you have a variable defined outside the object and you need to insert the value of that varaible as keyname into a object.
Example:
const key = "emai";
const person { name: "John", age: 30, hobby: ["hobby1", "hobby2"] }
person[key] = "email@gmail.com";

Iterating Objects

We can iterate objects using for in loop and using Object.keys.
Example:
const person { name: "John", age: 30, hobby: ["hobby1", "hobby2"] }

Using for in loop

```
for (let key in person) {
        console.log(person[key]);
}
```

Using Object.key

The Object.key method returns the object's key in the form of an array. So we can iterate object using for of loop.

```
for (let key of Object.keys(person)) {
        console.log(person[key]);
}
```

Computing Properties

Computed objects properties are used to take values of variablesfor the object keys from outside the object.

Example:

```
const key1 = "object1";
const key2 = "object2";

const value1 = "myvalue1";
const value2 = "myvalue2";

const obj = {
   [key1] : value1,
   [key2] : value2,
}
```

Without computed key we could have done it by creating an empty object and adding each key using bracket notation.

Example:

```
const obj1 = {};

obj1[key1] = value1;
obj1[key2] = value2;
```

Spread Operator in Objects

- Spread operator in objects works same as of the arrays. You can clone objects using spread operator, concat two objects as well as create objects from any iterable data types like strings, arrays, etc.
- While iterating the data types the index number of each element will be stored as the key of each value in object.

Example:

```
const obj1 = {
   key1 : "value1",
   key2 : "value2",
};
const obj12 = {
   key3 : "value3",
   key4 : "value4",
};
const newObj = {...obj1}; // cloning object
```

const newObj1 = {...obj1, ...obj12}; // concating two objects together

const newObj2 = {...obj1, key45 : "value45"} // adding extra key after obj1
const newObj3 = {..."abcd"}; // the index values are stored as the keys of the objects

Object Destructuring

- To destructure objects you need to specify the same value as object keys while defining variables.
  Example:
  const obj1 ={
  key1 : "value1", key2 : "value2",
  };
  const {key1, key2} = obj1;
  console.log(key1, key2);

- If you have more keys in objects then the defined variables, only the keys defined in variables will be assigned the associated key value.
  Example:
  const obj1 ={
  key1 : "value1", key2 : "value2", key3 : "value3"
  };
  const {key1, key3} = obj1;
  console.log(key1, key3);

- You can change the key name using colon notation while defining the variables.
  Example:
  const obj1 ={
  key1 : "value1", key2 : "value2",
  };
  const {key1 : var1, key2 : var2} = obj1;
  console.log(var1, var2);

- You can create saperate object of the key value pairs of the objects after defining few keys as variables. The saperate object will be created from all the key value pairs whose keys were not defined as variables.
  Example:
  const obj1 ={
  key1 : "value1", key2 : "value2", key3 : "value3", key4 : "value4", key5 : "value5"
  };
  const {key1, key3, …newobj} = obj1;
  console.log(key1, key3);

console.log(newobj) // This will contain key value pairs that are not defined as variables

Objects inside Array

You can create n number of objects inside arrays.

Example:

```
const array1 = [
   {userID : 1, userName : "username1", userGender : "male", userAge : 26},
   {userID : 2, userName : "username2", userGender : "female", userAge : 26},
   {userID : 3, userName : "username3", userGender : "male", userAge : 26},
];
```

Iterating arrays with objects

You can iterate arrays with objects just like the normal arrays using most commonly used loops in arrays (for of loop and traditional for loop).

Example using for of loop

```
for (const users of array1) {
   console.log(users);
}
```

Iterating objects inside arrays

You can iterate objects inside arrays using two for loops. One for iterating array and the other for iterating objects. You can use the commonly used loops for iteraring objects (for in loop and Objects.key method).

Example using Object.keys

```
for (const users of array1) {
   for (const user of Object.keys(users)) {
      console.log(`${user} : ${users[user]}`);
   }
}
```

Nested Destructuring

Nested destructuring means destructuring objects inside array. All the destructuring methods of objects works inside destructuring methods of arrays.

Reference array of objects

```
const array1 = [
   {userID : 1, userName : "username1", userGender : "male", userAge : 26},
   {userID : 2, userName : "username2", userGender : "female", userAge : 26},
   {userID : 3, userName : "username3", userGender : "male", userAge : 26},
];
```

Example of creating saperate objects from array of objects

const [user1, user2, user3] = array1;
console.log(user1);

Example of creating variables from object keys of the objects inside array
const [{userName}, , {userAge}] = array1;
console.log(userName);
console.log(userAge);

Example of changing the name of the variables of objects inside array
const[{userName : firstUser}, , {userName : thirdUser}] = array1;
console.log(firstUser);
console.log(thirdUser);

Functions in JavaScript

- Functions are used to do specific tasks by calling them.
- You can call a function multiple times in a program.
- There can be multiple functions in a program.

Function Declaration

You can declare a function using a function keyword and call the function using funtion name.
Example:

```
function printMessage() {
   console.log("Hello world from function");
};
printMessage();
```

- In functions, call function, invoke function, and run function means the same.
- Always create reusable functions by passing parameters.

Example:

```
function sumNumbers(number1, number2) {
   return number1 + number2;
};
const sumTwo = sumNumbers(4,6);
console.log(sumTwo);
```

Function Expressions

Function expression is another way to declare a function by assigning the function to a variable. Its recommended to use constant variable to declare create functon expression since the function is declared only once.
Function expression is also called as anonymous function.

Example 1:

```
const printMessage = function () {
   console.log("Hello world from function");
};

printMessage();
```

Example 2:

```
const sumNumbers = function (number1, number2) {
   return number1 + number2;
};
const sumTwo = sumNumbers(4,6);
console.log(sumTwo);
```

Arrow Functions

- We can create an arrow function by assigning the function to a variable and passing the parameters in rounded brackets followed by an arrow and curly parentheses.

  Example 1:

  ```
  const printMessage = () => {
     console.log("Hello world from function");
  };
  printMessage();
  ```

  Example 2:

  ```
  const sumNumbers = (number1, number2) => {
      return number1 + number2;
  };
  const sumTwo = sumNumbers(4,6);
  console.log(sumTwo);
  ```

- If we have a single statement of code written in a function, the curly parentheses and the return keyword inside the function can be skipped.

  Example:

  ```
  const sumNumbers = (number1, number2) => number1 + number2;

  const sumTwo = sumNumbers(4,6);
  console.log(sumTwo);
  ```

- If the function has only one parameter being passed, then we can skip the rounded brackets while passing the parameter.

  Example:

  ```
  const firstChar = myStr => myStr[0];
  const myString = "abcd";
  console.log(firstChar(myString));
  ```

Hoisting in JavaScript

- We can call a function before function declaration without any error and print a variable value before assigning a value to it this is called hoisting.
- Calling of function before works only in case of function declaration and not in cases of function expression and arrow function.
- Printing a variable value before assigning works only in case of variable created using var keyword and not by using let and const keyword.
- The value of the printed variable is undefined in case of printing a variable value before assigning using var keyword.

Functions inside Function

- A function can have multiple functions declared inside it.
- The functions inside the function can be declared using any of the function declaration methods like function declaration, function expression and arrow function.
- The functions declared inside a function have to be called from inside the function only.

Example:

```
const myFunc = () => {
   const myInsideFunc = () => {
      console.log("I am a function inside myFunc");
   }
   console.log("I am a myFunc function");
   myInsideFunc();
};
myFunc();
```

Lexical Scope

Lexical scope of a function means the environment in which a function is created/declared.

Example:

```
function myApp() {
   // this is lexical environment / scope of myFunc and myFunc2 funcions

   // const myVar = "value1";
   const myFunc = function() {
      // const myVar = "value24";
      console.log("Inside myFunc function ", myVar);
   };
   const myFunc2 = () => {};
   console.log(myVar);
   myFunc();
};
const myVar = "Value123";
```

myApp();

- In the above example the function myFunc will first check if the variable myVar is defined within its scope. If it finds the value it will print it.
- If it doesn't find the variable within its scope it will check if it is present in its lexical scope, which is in the scope of myApp function. If it finds the value it will print it.
- Still, if it doesn't find the variable myVar in its lexical scope it will further check in the lexical scope of MyApp function, which is the global scope. If it doesn't find the value in global scope it will throw an error.
- We also call this process lexical chaining.

Block scope vs Function scope

- Keywords let and const are block scope.
- Keyword var is function scope.
- A block in JavaScript is created using curly parentheses.
  Example:
  // block in JavaScript
  {
  // this is a block scope
  }

- If any block has variables defined using let and const, those variables can be accessed within that block. You cannot access let and const variables outside their block.
  Example:
  {
  let name1 = "name1";
  const name2 = "name2";
  console.log(name1, name2);
  }
  console.log(name1); // this will throw an error
  console.log(name2); // this will throw an error

- Variables defined using var can be accessed outside its block and also in other blocks within the same function. These variables be not accessible only when they are tried to from outside their function.
  Example:
  {
  var name3 = "myname";
  console.log(name3);
  }
  console.log(name3);
  {
  console.log(name3);
  }

Default Parameters

We can pass default parameters in the functions while defining them.
Default parameters are used in cases when the function takes multiple parameters but none or fewer parameters are passed.
Default parameters can help execute function code without errors.

Method 1 - Old codes

Using if statement inside the function.
Example:

```
function printNums(num1, num2) {
   // assigning default paramenter using if statement
   if (num1 === undefined && num2 === undefined) {
      num1 = 0;
      num2 = 0;
   } else if (num2 === undefined) {
      num2 = 0;
   }
   console.log(num1);
   console.log(num2);
}
printNums(12);
```

Method 2 - New

Assigning values at function declaration.
Example:

```
// assigning default paraments in function declaration
function printNums1(num1 = 0, num2 = 0) {
   console.log(num1);
   console.log(num2);
}
printNums1(2);
```

Rest Parameters

- If a function takes fewer parameters and we need to pass multiple parameters or we don't know the number of parameters we want to pass, then we use the rest parameter.
- The rest parameter stores all the parameters passed in an array. The rest parameter is created by adding three dots before the parameter.

Example:

```
function myFunc1(a, b, ...c) {
   console.log(a);
   console.log(b);
```

```
    console.log(c);
}
myFunc1(1,2,3,4,45, 5,6,87,9);
```

Parameter Destructuring

- Parameter destructuring is used with objects. We also use parameter destructing mostly in react framework.
- Parameter destructuring is used to pass object keys as the parameters in the function by destructuring an object.
- If we pass any key which is not present in an object, it will have a value of undefined.

Example:

```
const printObject = function ({name, gender, age}) {
    console.log(name);
    console.log(gender);
    console.log(age);
};
printObject(person);
```

Callback Functions

- Callback functions are the functions that take input as a function as its parameter.
- As a convention, we name the parameter as the ‘callback’ of the function that is taking a function as an input.
- We can name the parameter as anything we like but following convention is a good practice.
- Any function takes input as a function and calls it from inside it is a callback function.
- This is also called as a higher order function.

Example:

```
function myFunc1(name) {
    return `hello ${name}`;
};

function myFunc2(callback) {
    console.log(callback("user"));
    console.log("you are in myFunc2");
};
myFunc2(myFunc1);
```

Function returning Function

- Any function in JavaScript can have a function declared inside it and return that function.
- The variable assigned to the function call will be the function name of the returned function.

- This is a higher order function.
- Any function that takes input as a function or returns a function or does both is called a higher order function.

Example:

```
function myFunc() {
   function hello() {
      return "I am retuned from myFunc";
   }
   return hello;
};
const value = myFunc();
console.log(value());
```

Important Array Methods

forEach Method

- forEach array method takes an argument as a function (callback function).
- forEach method passes the array element and the index number from start to end as argument in callback function.
- In the callback function of the forEach method, you can either accept both array element and its index as parameters or either one of them as parameters.
- You can think forEach as a loop that returns all the elements and the indexes of an array.
- You can't return a value from the callback function in forEach method using return. But you can store the value by declaring a variable outside the method.

Example 1: Declaring function outside forEach method

```
const numbers = [4, 2, 6, 8, 5];
function displayArr(number, index) {
   console.log(`At ${index} the value is ${number}`);
};
numbers.forEach(displayArr);
```

In the above example we dont have to pass arguments inside function call. The forEach method will automatically pass the arguments.

Example 2: Declaring anonymous function inside forEach method

```
numbers.forEach(function(number) {
   console.log(number);
});
```

Example 3: Arrow function inside forEach method

```
numbers.forEach(number => {
   console.log(`${number} * 2 = ${number * 2}`);
});
```

map Method

- The map method takes an argument as a function (a callback function). You can either declare the function outside or you can declare an anonymous function or arrow function inside the map method.
- The map method is like forEach method but the only difference is the callback function passed inside the map method should return something.
- Like forEach method, map method passes array element and its index as arguments in the callback function.
- The map method returns an array of the values returned from the callback function inside it. If you don't return anything from the function then it will create an array of undefined values.

Example 1: declaring function outside

```
const myArray = [2, 4, 3, 6, 8, 12];

const square = function(number) {
   return number * number;
};
const squareArray = myArray.map(square);
console.log(squareArray);
```

Example 2: Using anonymous function

```
const squareArr = myArray.map(function(number, index) {
   return (`${index} : ${number * number}`);
});
console.log(squareArr);
```

Example 3: Using arrow function

```
const sqArray = myArray.map(number =>{
   return number * number;
});
console.log(sqArray);
```

filter Method

- The filter method takes an input argument as a function (callback function).
- The callback function of the filter method must always return a boolean value.
- The filter method will return an array of all the values of an original array which was returned as true from the callback function (returns array of truthy values).
- The callback function of the filter method can be declared outside the filter method or inside the method as an anonymous function or as an arrow function.

Example 1: declaring function outside

```
myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

```
const isEvenNumber = function(number) {
   return number % 2 === 0;
};
const evenNumArray = myArray.filter(isEvenNumber);
console.log(evenNumArray);
```

Example 2: Using anonymous function

```
const oddNumArray = myArray.filter(function(number) {
   return number % 2 !== 0;
});
console.log(oddNumArray);
```

Example 3: Using arrow function

```
const evenArray = myArray.filter((number) => {
   return number % 2 === 0;
});
console.log(evenArray);
```

reduce Method

- The reduce method takes input as a callback function like above 3 methods.
- The reduce method automatically passes the first array element as accumulator, the second array element as current value and the index of current value at the beginning.
- In the next iterations the accumulator takes the returned value from the first iteration and the current value and the index are incremented to the next element of an array.
- The reduce method returns the value of the accumulator at the end of the loop.
- In this case the loop runs (array-length - 1) times.

Example:

```
const numbers = [200, 3, 45, 50, 65];
// aim: sum of all numbers in array

const sum = numbers.reduce((accumulator, currentValue , index) => {
   console.log(`At ${index} iteration the value of accumulator: ${accumulator} the value of currentValue: ${currentValue} the return value: ${accumulator + currentValue}`);
   return accumulator + currentValue;
});
console.log(sum);
```

- To make the loop run the array-length times we can pass the accumulator value as 0 or number you wish in the reduce method as an input argument after the callback function.
- In this case the accumulator will take the value that is passed as argument, the current value will be the first element of an array and the index will also be of the first element of an array in the first iteration.

- The next iterations will continue taking returned value as accumulator until the end of the loop.

Example:

```
const numSum = numbers.reduce((accumulator, currentValue , index) => {
   console.log(`At ${index + 1} iteration the value of accumulator: ${accumulator} the value of currentValue: ${currentValue} the return value: ${accumulator + currentValue}`);
   return accumulator + currentValue;
}, 0);
console.log(numSum);
```

Note: Both the examples will give the same result but the first example will take less time since it runs for (array-length -1) times.

We need to pass the accumulator value when we are working with the real world arrays like array of multiple objects.

Example of a real world array

```
const myCart = [
   {productID : 1, productName : "Mobile", productPrice : 15000},
   {productID : 2, productName : "Watch", productPrice : 2000},
   {productID : 3, productName : "Laptop", productPrice : 50000},
   {productID : 4, productName : "TV", productPrice : 20000},
];

const totalPrice = myCart.reduce((currentPrice, {productPrice}, index) => {
   console.log(`At ${index +1 }, the current price = ${currentPrice} and the product price = ${productPrice} and total price = ${currentPrice + productPrice}`);
   return currentPrice + productPrice;
}, 0);
console.log(totalPrice);
```

sort Method

- Sorting means to arrange some items or things in a certain manner like assending order(low to high), descending order(high to low) etc.
- The sort method in array sorts the items in an array based on the ASCII values of the first character of each item in an array.
- The sort method does not returns a new array like forEach, map filter and reduce methods. It sorts the original array by mutatating/changing it.
- The sort array method first converts the array element to string, then finds the ascii value of the first character of the string and then sorts it accordingly.
- By default the sort method sorts items in ascending order (low to high).

Example 1: In case of numbers the output is not as per expected, since the numbers are sorted according to the ascii value of the first digit.

```
const numbers = [5, 9, 1210, 400, 3000, 1209];
```

```
numbers.sort();
console.log(numbers);
```

Example 2: In case of strings the result is as per expected

```
const userNames = ["lelwyn", "tim", "arun", "kunal", "cody"];
userNames.sort();
console.log(userNames);
```

- To get the desired output (ascending order) in case of numbers we pass an optional callback function as input in the sort method.
- The callback function takes two elements(a, b) from the array in each iteration, subtract them(a-b) and if the result is positive, the elements are swapped in an array and if the result is negative, the elements are kept as they are.
- This method of passing callback function does not apply to strings and character elements and give unexpected output.

Example:

```
numbers1.sort((a,b) => {
   console.log(`the value of a is ${a} the value of b is ${b} and the result is ${a - b}`);
   return a - b;
});
console.log(numbers1);
```

- To sort the numbers in descending order, we pass the same callback function with slight modification.
- We subtract the second element from first element(b-a).

Example:

```
const numbers2 = [5, 9, 1210, 400, 3000, 1209];
numbers2.sort((a,b) => {
   console.log(`the value of a is ${a} the value of b is ${b} and the result is ${b - a}`);
   return b - a;
});
console.log(numbers2);
```

find Method

- The find method takes input as callback function.
- The find method returns the first occurance of the array element which is returned as truthy value by the callback function passed inside it as argument.
- If the callback function never return any thuthy value then then find method will return undefined.

Example:

```
const lengthThree = animals.find((element) => element.length === 3);
console.log(lengthThree);
```

every Method

- The every method takes input as a callback function which returns a boolean value.
- The every method also return boolean value. It returns truthy value only when the callback function returns truthy values for all the iterations.
- The every method terminates and retuns falsy value as soon as the callback function return a falsy value.

Example:

```
const numbers = [2, 4, 6, 8, 10, 12];

const checkEven = numbers.every((number) => number % 2 === 0);
console.log(checkEven);
```

some Method

- The some method takes input as a callback function that retuns a boolean value.
- The some method also returns a boolean value. It returns truthy value when the callback function returns truthy value atleast once.
- The some method terminates and returns truthy value as soon as the callback function return a truthy value.

Example:

```
const numbers = [1, 3, 5, 7, 4, 11];

const findEven = numbers.some((number) => number % 2 === 0);
console.log(findEven);
```

fill Method

- The fill method takes input as 3 arguments, the value, the start and the end.
- The value is the element that will be filled in an array. The start is the index number from where the value should be filled. The end is the index number before which the filling of value shoud be stopped.
- The fill method has two use cases.
- **Case 1:** To create a new array of a specific length filled with same elements.
    - To create a new array we use array constructor and only specify the value in the fill method since there is no start and end value in empty array.

      Example: creates an array of length 10 filled with -1

      ```
      const myArray = new Array(10).fill(-1);
      console.log(myArray);
      ```

- **Case 2 :** To change certain elements from an existing array.
    - Here the fill method changes the original array and does not create any separate array.

- The fill method takes the value to be fill first, then the starting index number and then the ending index number.
  Example:
  const numbers = [1, 2, 3, 4, 5, 6, 7, 8];
  numbers.fill(0, 2, 5);
  console.log(numbers);

splice Method

- The splice method is used to delete array elements or insert element in an array or do both to an array.
- The splice method takes 3 arguments as input, the start value, the delete value, and the insert value.
- The start is the index of an array from where deletion or insertion will start. The delete value is the number of elements that will be deleted from the index number in start. The insert values are the values that will be inserted into an array from the index number in start.
- During insertion and deletion the splice method changes the original array but returns an separete array of all the delected items.

Example: Delete Items
If you only want to delete the insert value is not necessary.

```
const itemArray1 = ["item1", "item2", "item3", "item4", "item5"];
const delArray1 = itemArray1.splice(2, 2);
console.log("original array: ", itemArray1);
console.log("deleted: ", delArray1);
```

Example: Insert Items
If you only want to insert the delete value must be 0 before inserting items.

```
const itemArray2 = ["item1", "item2", "item3", "item4", "item5"];
itemArray2.splice(4, 0, "newItem1", "newItem2",);
console.log("original array: ", itemArray2);
```

Example: Both Insert and Delete Items

```
const itemArray3 = ["item1", "item2", "item3", "item4", "item5"];
const delArray3 = itemArray3.splice(1, 3, "newItem1", "newItem2");
console.log("original array: ", itemArray3);
console.log("deleted: ", delArray3);
```

Iterables and Array-like Objects

Iterables

- Iterables are those on which we can apply for of loop.
- Strings and Arrays are iterables.
- Object is not iterable.

Example: String

```
const firstname = "Lelwyn";
for (const char of firstname) {
   console.log(char);
}
```

Example: Array

```
const numbers = [1, 2, 3, 4, 5];
for (const number of numbers) {
   console.log(number);
}
```

Array-like Objects

Array-like objects are those which length property and can be accesed through their index numbers.
Strings is an example of Array-like object.

Example:

```
const myName = "lelwyn vaz";
console.log(myName.length);
console.log(myName[4]);
```

Sets

- Sets are iterables and are used to store only iterables type data linearly like arrays.
- Sets also have their own methods.
- You can only store unique items in sets, no duplicate items are allowed.
- Sets does not have index based access, hence the order of items is not guaranteed.

Example: Creating a set

```
const numbers = new Set([1, 2, 3, 4]);
console.log(numbers);
```

Example: duplicates are not added even if we specify

```
const numbers1 = new Set([1, 1, 2, 3, 3, 4]);
console.log(numbers1);
```

Example: Common way of creating and adding items to a set

```
const items = ["item1", "item2"];
const mySet = new Set();
mySet.add("abc");  // can add items used add method
mySet.add([2, 4, 8]); // can add different types of items
mySet.add(1);
mySet.add(items);
mySet.add(items); // duplicate is not added
mySet.add(["item3", "item4"]);
mySet.add(["item3", "item4"]); // this is added, because this is saperate array not a duplicate
console.log(mySet);
```

Set methods

- add Method: It is used to add items to the set.
  Example:

  ```
  const mySet = new Set();
  mySet.add("abc");
  console.log(mySet);
  ```

- has Method: It is used to check whether an item is present in a set or not.
  Example:

  ```
  console.log("1 present? ", mySet.has(1));
  ```

Example: Using for of loop on set.

```
for (const item of mySet) {
   console.log(item);
}
```

Example: Finding length of a set
Cannot use length property on sets

```
console.log(uniqueElements.length); // cannot use (undefined)
let setLen = 0;
for (const element of uniqueElements) {
   setLen += 1;
}
console.log("the length of uniqueElements: ", setLen);
```

Maps

- Maps stores data in key value pairs and no duplicate keys as allowed just like the objetcs.

- Map is an iterable.
- Maps also stores data in ordered fashioned.

Difference between Maps and Objects

- Objects can only have string or symbol as key.
- In maps you can use anything as key like array, string, number, etc.
- You cannot use for of loop in objects but you can use for in loop.
- You can use for of loop in Maps.

Creating a map

Example: Method 1

```
const person = new Map();
person.set('firstName', "lelwyn");
person.set('lastName', "vaz");
person.set(1, "number");
person.set([1, 2, 3], "key can be anything");
person.set({"other details" : "More personal info"}, {"age" : 26, "gender" : "male"});
console.log(person);
```

Example: Method 2

```
const otherPerson = new Map([['firstname', 'cody'],['lastname', 'greay'], [24, 'number']]);
console.log(otherPerson);
```

Getting items from Map

```
console.log(person.get('firstName'));
console.log(person.keys());
for (const key of person.keys()) {
   console.log(key, typeof key, );
};
```

Using for of loop

```
for (const items of person) {
   console.log(items, Array.isArray(items));
};

for (const [key, value] of person) {
   console.log(key, "-->", value);
};
```

Clonning Objects and Optional Chaining in Objects

Cloning using Object.assign

- The object.assign method is used to clone an original object in the heap memory to a new object.
- The cloned object will be stored saperately and will different reference address which means, if the original object is changed after cloning, the cloned object will not be changed or affected.
- The object.assign method was used mostly in older code bases. Currently there is a better alternative to clone, which is using the spread operator.

Example:

```
const myObj1 = {
   key1: "value1",
   key2: "value2"
};

const obj1Copy = Object.assign({}, myObj1);
console.log("obj1copy", obj1Copy);
myObj1.key4 = "value4";
console.log(myObj1);
console.log(obj1Copy);
```

Clonning using the spread operator

The spead operator is the modern way of clonning an object. It works same as the Object.assign method internally.

Example:

```
const newObj1 = {...myObj1};
console.log("newObj1", newObj1);
myObj1.key5 = "value5";
console.log(myObj1);
console.log(newObj1);
```

Optional Channing in Objects

- The optional channing operator is used when we need the code to run without errors even if the object or some of the object keys are currently not available.
- This is widely used in real world case such as react framework.
- The optional channing operator ‘?’ checks if the object or key is currently present or not. If present the key is returned. Else, if the object or key is not present it returns the value as undefined without throwing an error.

Example:

```
let users;
const obj1 = {
   key1: "value",
   key2: {key21: "value21", key22: "value22"},
};
console.log(obj1?.key2?.key21);
console.log(obj1?.key3);
console.log(obj1?.key3?.key31); // error will not appear now
console.log(users?.key1); //error will not appear now
```

Own Methods

A method is any function that is inside an object.

Creating Methods

- The object key which has the value a function becomes the function name.
- We can call the function/method using the key assigned to it.
- We can create methods in two ways
- We use this keyword to access the other keys within that object.

Method 1: Declaring the function inside the object

- We can declare function inside the object as the value of an objects key.
- This is limited to the same object unless you call the function/method using the call method.

  Example:

```
const simpleDemo = {
   firstname: "person1",
   age: 23,
   about: function() {
      // to get values of other keys in object, we use this keyword
      console.log(`the firstname is ${this.firstname} and the age is
${this.age}`);
   },
};
console.log(simpleDemo.about); // this will display the whole function
simpleDemo.about();
```

Method 2: Declaring the function outside the object

- We can declare a function outside an object and assign the function name to a key of an object.
- Methods declared outside can be accessed by and useful to multiple objects.
  Example:

```
function personInfo() {
   console.log(`person name is ${this.firstname} and the age is ${this.age}`);
};
personInfo(); // the values will be undefined

const person1 = {
   firstname: "Ielwyn",
   age: 24,
   about: personInfo,
};
const person2 = {
   firstname: "tim",
   age: 42,
   about: personInfo,
};
person1.about(); //now the this kwyword inside the personInfo method will have the value of person1
person2.about();
```

This Keyword

- We get to know the value of this when the code is running(during runtime) and not when the code is being written(compile time).
- The this keyword take the value of an object as a whole when the methods are being called. So we can refer this keyword as the current object that is calling the method
- By default in the global scope this keyword the value of window object. In JavaScript the window is the global object.
  Example:

```
console.log(this);
console.log(window);
console.log(this === window);
```

- The this keyword inside normal function will have a value of window object since the function is defined in the global scope which is the scope of window object.
  Example:

```
function demoFunction() {
   console.log(this);
   console.log(this === window);
```

```
};
demoFunction();
window.demoFunction() // same as demoFunction()
```

- You can find the function defined in global scope inside the window object if you print the window object using window keyword.
- To avoid geting the window object in the function scope we can use the strict mode inside the function or in the global scope of the JavaScript file.
  Example:

```
"use strict" // you can define it on top or indise the function

function demoFunction1() {
   "use strict"
   console.log(this);
   console.log(this === window);
};
demoFunction1();
```

call Method

- The call method is use to call the functions like the normal function call. But it works differently in methods.
- In normal functions it works just like the normal function call.
- The call method works perfectly fine in normal functions because it takes default argument as window object.
  Example:

```
function hello() {
   console.log("hello world");
   console.log(this);
};
hello();
hello.call(); // we can also call normal functions using call method
```

- When we need to call methods or functions inside objects, we have to pass the argument as the current object to which the this keyword will be referring to in the call method.
- If we don't pass any argument by default the call method will refer this keyword to window object.
  Example:

```
const user1 = {
   firstname: "lelwyn",
   age: 25,
   about: function() {
      console.log(this);
```

```
        console.log(this.firstname, this.age);
    },
};
const user2 = {
    firstname: "bob",
    age: 35,
};

user1.about(); // normal method call
user1.about.call(); // here the value of this keyword will be window object by default
user1.about.call(user1);
user1.about.call(user2);
```

- If the methods or functions inside objects has any parameters, we can pass them as arguments in the call method.
  Example:
```
const user1 = {
    firstname: "lelwyn",
    age: 25,
    hobbies: function (hobby1, hobby2) {
        console.log(`hobbies of ${this.firstname} are: ${hobby1}, ${hobby2}`);
    },
};
user1.hobbies.call(user1, "drawing", "music"); //passing multiple arguments
user1.hobbies.call(user2, "playing", "music");
```

- If we have methods declared outside the objects and assigned the value of the objects key to the methods names, then we can call the functions by just function name and passing the object inside the call method.
  Example:
```
const user1 = {
    firstname: "lelwyn",
    age: 25,
};
const user2 = {
    firstname: "bob",
    age: 35,
};
function knownLang(lang1, lang2) {
    console.log(`${this.firstname} can code in ${lang1} and ${lang2}`);
};
knownLang.call(user1, "python", "JavaScript");
knownLang.call(user2, "python", "TypeScript");
```

apply Method

The apply method works same as the call method internally but, takes the method arguments as an array of arguments.

Example:

knownLang.apply(user1, ["python", "JavaScript"]);
knownLang.apply(user2, ["python", "TypeScript"]);

bind Method

- The bind method return the method/function inside the object as a function.
- The bind method is useful in cloning the methods ouside objects.
  Example:
  const myFunc1 =knownLang.bind(user1, "python", "JavaScript");
  const myFunc2 =knownLang.bind(user2, "python", "TypeScript");
  myFunc1();
  myFunc2();

Small warning/Note

- When you want to clone a method or function inside the object outside that object make sure you are using the bind method to clone it.
- If not cloned using bind method the this keyword inside the cloned function will represent to window object and not to the parent object of that method.

Example:

```
const myObj = {
   key1: "lelwyn",
   key2: 29,
   about: function() {
      console.log(this);
      console.log(`name: ${this.key1} and age: ${this.key2}`);
   },
};
const myFunc = myObj.about; // this keyword will still represent windows object
myFunc();

// always clone functions using bind method
const correctFunc = myObj.about.bind(myObj);
correctFunc();
```

Arrow Function and this keyword

- The arrow methods behave differently with this keyword from other traditional methods.
- The arrow function inside object or arrow method does not have its own this.

- The arrow method takes the value of this keyword from its surroundings that is one level up which is window object.
- You also cannot change the value of this keyword in arrow methods using call method while calling the method. It will still take the value of window object.

Example:

```
const myObj = {
   key1: "lelwyn",
   key2: 29,
   about: () => {
      console.log(this);
      console.log(`name: ${this.key1} and age: ${this.key2}`);
   },
};
// the this of arrow function will be one level up
myObj.about();
// you cannot change this of arrow function
myObj.about.call(myObj); // the object at 1 level up is the window object
```

Short Syntax for Methods

While creating methods(functions inside objects) inside the objects you don't need to write the key value pair.
You can directly write function name without function keyword as key.

Example:

```
const myObj1 = {
   key1: "lelwyn",
   key2: 29,
   about() {
      console.log(`name: ${this.key1} and age: ${this.key2}`);
   },
};
myObj1.about();
```

Object Oriented JavaScript

Functions to create multiple Objects

- If we have a real world data of multiple users and we need a saperate object for each user then it would be tedious to create each object separately.
- We can create such objects using functions.
- We start with creating a function that takes multiple inputs and creates an object with key value pair where the keys are taken from the inputs.

- We can also create and assign methods to the object if needed and at last the function will return an object.
- The drawback in this approach is that the same methods are created multiple times which is equal to the number of objects.
- The functions that are used to create an object or multiple objects are called constructor functions.

Example:

```
const createUsers = function (firstname, lastname, email, age, address) {
   const user = {};
   user.firstName = firstname;
   user.lastName = lastname;
   user.email = email;
   user.age = age;
   user.address = address;
   user.about = function () {
      return `${this.firstName} is ${this.age} years old.`;
   };
   user.is18 = function () {
      return this.age >= 18;
   };
   return user;
};
const newUser = createUsers("lelwyn", "vaz", "email@gmail.com", 49, "my address");
console.log(newUser);
console.log(newUser.about());
```

We know that method can be defined outside the object and same method can be used by multiple objects because the this keyword takes its value during runtime.

Store methods in different Object

- We can store all the same methods in different object and assign the reference of the object where the methods are stored to the methods of the object which is being created.
- This way we create methods only once and use them multiple times saving the memory used to store the code.

Example:

```
const userMethods = {
   about: function () {
      return `${this.firstName} is ${this.age} years old.`;
   },
   is18: function () {
      return this.age >= 18;
   },
```

```
};


const createUsers = function (firstname, lastname, email, age, address) {
   const user = {};
   user.firstName = firstname;
   user.lastName = lastname;
   user.email = email;
   user.age = age;
   user.address = address;
   user.about = userMethods.about; // refering by address
   user.is18 = userMethods.is18;
   return user;
};

const newUser1 = createUsers("lelwyn", "vaz", "email@gmail.com", 49, "my address");
const newUser3 = createUsers("tim", "smith", "email@gmail.com", 17, "my address");
console.log(newUser1.about());
console.log(newUser3.is18());
```

The drawback in this case is when we have multiple methods in the methods object and we need to assign all those methods to the methods of the objects that is being created.

Creating multiple objects using object.create

- We can create the object that creates multiple objects using the object.create method and pass the input as the object with all the methods.
- This way we dont have to create or refer the address of the object methods again in the object that creates multiple objects inside the function.
- This is the best approach to create multiple objects.

Example:

```
const userMethods = {
   about: function () {
      return `${this.firstName} is ${this.age} years old.`;
   },
   is18: function () {
      return this.age >= 18;
   },
};


const createUsers = function (firstname, lastname, email, age, address) {
   const user = Object.create(userMethods); // prototype chaining
   user.firstName = firstname;
```

```
    user.lastName = lastname;
    user.email = email;
    user.age = age;
    user.address = address;
    return user;
};
const newUser1 = createUsers("lelwyn", "vaz", "email@gmail.com", 49, "my address");
const newUser3 = createUsers("tim", "smith", "email@gmail.com", 17, "my address");
console.log(newUser1.about());
console.log(newUser3.is18());
```

Into to Object.create() Method

- We can also create an empty object using the object.create method, the create method take the input as an object.
- This input object is set as the proto/prototype of that object which means, if any of the key is not in the current object, then JavaScript will first look into the proto object before assigning the value as undefined.
- In the official EcmaScript/JavaScript documentation the object proto is written in [[prototype]] format and some browsers use __proto__ (dunder proto) format.
- So the [[prototype]] and __proto__ are the same. We can display the proto objects details in console using the dunder proto method.
- In simple language the proto is used to set the prototype chaining between multiple objects or between object and constructor function prototype.

Example:

```
// creating demo object to access values
const myobj1 = {
    key1: "value1",
    key2: "value2",
};

// creating object using Object.create()
const myobj2 = Object.create(myobj1);
myobj2.key3 = "value3";
myobj2.key4 = "value4";
console.log(myobj2);
console.log(myobj2.__proto__);
console.log(myobj2.key3);
console.log(myobj2.key2);
```

Prototype - (Only available in Functions)

- In JavaScript functions are treated as the functions as well as the objects. So we can say functions are same as objects.

- We can add our own properties to the functions as key value pairs like in objects.
- Functions also provide us with call, apply and bind methods and also some more useful properties.
- One of the property is name which tells the name of the function.

Example:

```
const hello = function() {
   console.log("hello world");
};
hello();
console.log(hello.name); // name property tells function name
// adding own properties
hello.myOwnProperty = "unique value";
hello.myFunction = function() {
   console.log("this is unique function");
};

console.log(hello.myOwnProperty);
hello.myFunction();
```

- Functions provide us with a free space that is an empty object and that object is called a prototype of a function.
- We can access the prototype of the function using the prototype property of the function.
  Example:
  ```
  console.log(hello.prototype);
  ```

- The prototype property is available only in the functions and that is how we can differentiate functions with objects.
  Example:
  ```
  const hello1 = {key1: "value1"};

  if (hello1.prototype) {
     console.log("prototype present");
  } else {
     console.log("prototype not present");
  }
  ```

- Since the prototype of the functions is an empty object we can perform all the operations of a normal object like adding key value pairs etc.
  Example:
  ```
  hello.prototype.key1 = "value1";
  hello.prototype.demoMethod = function() {
     return "I am demo method";
  };
  ```

console.log(hello.prototype);
console.log(hello.prototype.demoMethod());

Difference between proto and prototype

- Proto (__proto __) is a property of an object that points to its constructor's prototype or some other object declared outside, allowing the object to inherit properties and methods.
- Prototype is a property of a constructor function used to set properties and methods that will be shared by all instances or objects.
- In simple terms proto the property of an object which is use to inherit some other objects properties and methods or inherit properties or methods from the prototype of a function and proto is found on every object that is created.
- On the other hand, the prototype is the empty object created every time a function is created and is used to define shared properties and methods for all the instances/objects of the constructor function.

Use of prototype to create multiple objects

- We can create multiple objects using the prototype property of the function.
- We can create a function that creates and returns multiple objects.
- The prototype of the function that creates objects will have all the common methods and properties shared by the objects.
- Then we can point the proto of those objects to the prototype of the function that creates those objects.
- This way we don't need to create separate object to store the methods of the objects being created.

Example:

```
const createUsers = function (firstname, lastname, email, age, address) {
   const user = Object.create(createUsers.prototype); // using prototype of the function to inherit
   user.firstName = firstname;
   user.lastName = lastname;
   user.email = email;
   user.age = age;
   user.address = address;
   return user;
};

createUsers.prototype.about = function () {
   return `${this.firstName} is ${this.age} years old.`;
};

createUsers.prototype.is18 = function () {
```

```
    return this.age >= 18;
};

const newUser1 = createUsers("lelwyn", "vaz", "email@gmail.com", 49, "my address");
const newUser2 = createUsers("harshit", "vashisth", "email@gmail.com", 23, "my
address");
const newUser3 = createUsers("tim", "smith", "email@gmail.com", 17, "my address");
console.log(newUser1);
console.log(newUser2);
console.log(newUser1.about());
console.log(newUser3.is18());
```

New keyword

The new key does three things:

- It creates an empty object with the help of the constructor function.
- It sets the value of this keyword to the current empty object created and returns the object.
- It points the proto of the object created to the prototype of the constructor function.

As a convention the functions which will be called using the new keyword, the name of those functions starts with capital letter for better understanding of code.

Example:

```
const CreateUsers = function (firstname, lastname, email, age, address) {
    this.firstName = firstname;
    this.lastName = lastname;
    this.email = email;
    this.age = age;
    this.address = address;
};

CreateUsers.prototype.about = function () {
    return `${this.firstName} is ${this.age} years old.`;
};

CreateUsers.prototype.is18 = function () {
    return this.age >= 18;
};
// calling using new keyword
const newUser1 = new CreateUsers("lelwyn", "vaz", "email@gmail.com", 49, "my
address");
console.log(newUser1);
```

The hasOwnProperty Method

- If we run a for in loop on an object it will return all the keys of the object including the key of the proto.
- If we want only the keys owned by the object then we can use the hasOwnProperty method and pass each key inside the method. It wil return a boolean value.

Example:

```
for (const key in person) {
   if (person.hasOwnProperty(key)) {
      console.log(key);
   }
};
```

More about prototype

- We know that only the functions has the prototype.
- But in JavaScript internally all objects that can be created using the new keyword the predefined prototype.
- When we create objects like arrays, objects, etc JavaScript internally creates those objects using the objects constructors.
- And we know that the constructors are the functions that are used to create objects. So, each object being created inherits the inbuilt prototype of its constructor.
- Because of the constructor prototype properties we get to use all the available inbuilt methods of each objects.
- You can find all the prototype properties of the object in its proto as we know the new keyword points the proto to the prototype of the constructor function.

There are 2 ways to view the properties of an object.

Example of an array object:

```
const myArr = [1, 2, 3];
```

To view the prototype properties we use

Method 1

```
console.log(Object.getPrototypeOf(myArr));
```

Method 2

```
console.log(myArr.__proto__);
```

In case of array the prototype object is converted array since array is also an object

```
function arrConstructor () {
   console.log("demo");
};
console.log(arrConstructor.prototype);
arrConstructor.prototype = [];
console.log(arrConstructor.prototype);
```

Class keyword

- The class keyword creates classes just like in other programing languages which are used to define the properties and methods at one place.
- But the classes in JavaScript are fake and the class keyword internally works just like creating multiple objects using the new keyword.

Example:

```
class CreateUsers {
   constructor(firstname, lastname, email, age, address) {
      this.firstName = firstname;
      this.lastName = lastname;
      this.email = email;
      this.age = age;
      this.address = address;
   };
   about() {
      return `${this.firstName} is ${this.age} years old.`;
   };
   is18() {
      return this.age >= 18;
   };
};

const newUser1 = new CreateUsers("lelwyn", "vaz", "email@gmail.com", 49, "my
address");
console.log(newUser1);
console.log(newUser1.about());
```

Extends keyword

- The extends keyword is used in class declarations to create a subclass (child class) that inherits properties and methods from a parent class.
- It establishes a relationship between classes, where the child class can reuse and extend the behavior of the parent class.
- The extends keyword only inherits the non static properties and methods from a parent class.

Example:

```
class Animal {
   constructor(name, age) {
      this.name = name;
      this.age = age;
   };

   eat() {
```

```
    return `${this.name} is eating`;
  };
};
class Domestic extends Animal {
  isDomentic() {
    return `${this.name} is domestic animal`;
  };
};

const dog = new Domestic ("tommy", 2);
console.log(dog.eat());
console.log(dog.isDomentic());
```

Super keyword

- The super keyword is used inside the child class to access/call the properties and methods of the parent class.
- It's commonly used in constructors to call the parent class constructor to initialize the parent and child class properties properly.
- It can also be used to call overridden(same methods in child class) methods from the parent class.

Example:

```
class Domestic extends Animal {
  constructor(name, age, speed) {
    super(name, age);
    this.speed = speed;
  };

  runs() {
    return `${this.name} can run at the speed of ${this.speed}kmph`;
  };
};

const dog = new Domestic ("tommy", 2, 30);
console.log(dog);
console.log(dog.runs());
```

Same method in parent class and child class

- If there is the same method in the parent class and the child class, and if the method is called using the child class object/instance the method in the child class will be executed.
- It is because JavaScript first looks for the methods in the object/instance itself and then looks into the parent class methods.

- This behavior is called overriding in JavaScript.
- If we still want the parent class method to be executed in child class we can make use of super keyword.
- If the method is called using the parent class object/insance, then the parent class method will be executed.
- The object created using the parent class will not have access to the child class methods because the parent class does not inherit properties and methods of child class.

Example:

```
class Domestic extends Animal {
   eat() {
      return `${this.name} eats food at home when given.`;
   }

   combineEat() {
      return `${super.eat()} and ${this.eat()}`;
   }
};

const dog = new Domestic ("tommy", 2);
console.log(dog.eat());
console.log(dog.combineEat());
```

Getters and Setters

The objects methods are either getters or setters.

Getters

- Getters are the methods that allow you to access the value of an objects properties.
- Getters are basically used to retrieve the value of an objects properties. The retrived value may involve some calculations and transformations before returning.
- Any objects method that retuns some value is a getter.
- A method that is set as a getter becomes the property of an object and you can access the getter method like a property of an object and not as the function call.

Setters

- Setters are the methods that allow you to modify the value of an objects properties.
- Setters are basically use to change or set the value of an objects properties in a controlled manner. There may involve some validations and adjustments before assigning the value.
- Any objects method that can change the value of an objects property is a setter.
- The setter method must have atleast one parameter while defining and takes atleast one input during the method call.

- Setters also are properties of an object and you can set the new value like changing the value of an objects property.

Example: getters and setter

```
class person {
   constructor(firstname, lastname, age) {
      this.firstname = firstname;
      this.lastname = lastname;
      this.age = age;
   };

   get fullname() {
      return `${this.firstname} ${this.lastname}`;
   };
   set fullname(name) {
      const [firstname, lastname] = name.split(' ');
      this.firstname = firstname;
      this.lastname = lastname;
   };
};

const user = new person("tim", "gray", 36);
console.log(user);
console.log(user.fullname); // getting fullname using getter
user.fullname = "lelwyn vaz"; // setting name using setter
console.log(user.fullname);
```

Static methods and properties

- The methods and properties that has static keyword before declaration in the class cannot be accessed by an object of that class or by an object of its child class.
- Static methods and properties can be accessed only by the class itself and the child class.
- Static methods are mostly used in large projects when we need to initialize the apps or modules without creating any object of that class.

Example:

```
class animal {
   constructor(name, age) {
      this.name = name;
      this.age = age;
   };
   static animalType = "wild and domestic";
   static classInfo() {
      return "this class creates animal object";
```

```
    };
};

class dog extends animal {
    get isDomestic() {
        return `${this.name} is a domestic animal`;
    };
};

const wild = new animal("lion", 20);
const domestic = new dog("tommy", 5);
// console.log(wild.classInfo()); // error
console.log(domestic.animalType); // undefined value
console.log(dog.classInfo());
console.log(animal.animalType);
```