Data types

① Call by value (Primitive) →
(i) String
(ii) Number
(iii) Boolean
(iv) null
(v) undefined
(vi) Symbol
(vii) BigInt

② Call by Reference (Non-Primitive)
(i) Arrays
(ii) Objects
(iii) Functions

JavaScript is a dynamically typed language. This means variable types are determined at runtime and you don't need to explicitly declare the type of variable before using it.

→ Primitive data types used stack memory.
→ Non-Primitive data types used heap memory.

☆ Creating object →

```
let obj = {
    name: "Vasu",
    Age : 21
}
```

- To access data from objects →

obj.name

Strings

(i) concactenation →

method 1 →

```
const name = "hitesh"
const repoCount = 50

console.log(name + repoCount + "value")

//This is not a good practise
```

method 2 →

```
console.log(`Hello my name is ${name} and
my repo count is ${repoCount}`);
```

(ii) another way to declare string →

```
const gameName = new String('vasu')
```

now this string is in form of key value pair. i.e object.

```
0: "v"
1: "a"
2: "s"
3: "u"
```

The advantage of declaring string in this way is that we get various methods and functions to work with.
you can see these methods in console of your browser.

Some methods are →

(i) console.log(gameName.__proto__);
Return empty object → {}

(ii) console.log(gameName.length)

(iii) console.log(gameName.toUpperCase());

(iv) console.log(gameName.charAt(2));

(v) console.log(gameName.indexOf('t'));

(vi) const newString = gameName.substring(0, 4)
→ The 4th index element will not get included.

(vii) gameName.slice(0, 4)
→ same as substring but we can give negative values in it to print reverse string

(viii) const newString = " hitesh "
console.log(newString.trim());

→ This function will remove every extra space.

(ix) const url = "https://hitesh.com/hitesh%20choudhary"
console.log(url.replace('%20', '-'))
what value to replace — which value to replace with

(x) console.log(url.includes('sundar'))
→ This will check weather this string have that particular word or not.

Search more about functions from JS mdn web docs.

Numbers and Maths

const score = 400
const balance = new Number(100)

The difference in above two statements is sameas that of string.

The various functions are→

(i) balance.toString()
→ This will convert the number into string and you can now also use string functions with it.

(ii) balance.toFixed(1) →
In this we pass a number and based on that number the decimal and zeros will get added.
eg→ const balance = new Number(100)
console.log(balance.toFixed(1))

Output→ 100.0

(iii) const otherNumber = 123.8966
console.log (otherNumber.toPrecision(4));
↓
The Range is between 1-21

→ This method formats a number to a specific length.
→ A decimal point and nulls are added (if needed), to create the specified length.

Output → 123.9

→ Suppose number is same i.e 123.8966 but you gave 2 in precision, then it will do the job but will add rest of the number in the form of exponensial formatoon and output will be 1.2e+2

(iv) const hundreds = 1000000
console.log (hundreds.toLocaleString());

Output → 1,000,000

→ Format a number into a string, using locale settings.
→ Read more about this function using W3 Schools.

(v) console.log (Math.abs(-4));
→ This function will give absolute value of a number.

Output → 4

4 ⟶ 4 $\qquad$ −4 ⟶ 4

(vi) console.log(Math.round(4.3))

→ This will round off the number that you had given.

Output → 4

eg: 4.3 → 4 and 4.6 → 5

(vi) console.log(Math.ceil(4.2))

→ This function will give ceiling of a number.

Output → 5

(vii) console.log(Math.floor(4.6))

→ This function will give floor of a number.

Output → 4

(viii) console.log(Math.min(4, 3, 6, 8));

→ This will give minnimum value from an array.

Output → 3

(ix) console.log(Math.max(4, 7, 9, 10))

→ This will give maximum value from an array.

Output → 10

(x) console.log(Math.random())
→ This will give random value between 0 and 1.

** Note →
Now suppose we want the value between some specific value for that →

```
const min = 10
const max = 20

console.log(Math.floor(Math.random()*(max-min+1))
            +min)
```

To know more about this check 12th video of chai aur Javascript playlist of channel chai aur code.

Date and Time

Note → The date function will be modified and will become easy to use in few years. So stay up to date about it on Javascript mdn.
we will learn Date and time in traditional format for now.

(i) let myDate = new Date()
console.log(myDate)

Output → 2024-03-14T12:19:04.288Z
→ This is not much readable

To make this much readable we need to use some functions.

→ console.log(myDate.toString())
output ⇒ Thu Mar 14 2024 17:51:34 GMT+0530

By this you will get output in much readable format.
There are some more similar functions that you can try and observe the output.

→ Date is of type object in JavaScript.

creating own date →
let myCreatedDate = new Date(2023, 0, 23)
↓
month start from 0 in Java Script

console.log(myCreatedDate.toDateString());

Output → Mon Jan 23 2023

To get time also →

let myCreatedDate = new Date(2023, 0, 23, 5, 3)
console.log(myCreatedDate.toLocaleString());

Output → 1/23/2023, 5:03:00 AM

To get in specific format
let myCreatedDate = new Date("2023-01-14")
console.log(myCreatedDate.toLocaleString());

Time Stamp

The data is from January 1, 1970 midnight. That is if we get time of present date the output will be in milliseconds calculated from January 1, 1970.

eg→

```
let myTimeStamp = Date.now() //get tody date
console.log(myTimestamp);
```

Output→ 1677672589191

Now to get output of specific date →

→ let myCreatedDate = new Date("01-14-2023")
console.log(myCreatedDate.getTime());

Output→ 1673654400000

Arrays

```
const myArr = [0, 1, 2, 3, 4, true, "hello"]
```

→ Elements in the array can be of same type and as well of different types.
→ Javascript arrays are resizable
→ In java script there are no associative array and you can access the element in array in following way:

```
console.log(myArr[0]);
```

Output → 0

→ Basically arrays Element is accessed using index value.
→ The indexing is zero based.

Another method →

```
const myArr2 = new Array(1, 2, 3, 4)
```

Array methods

```
const myArr = [1, 2, 3, 4]
```

(i)
```
myArr.push(6)
console.log(myArr);
```

Output → [1, 2, 3, 4, 6]

(ii)
```
myArr.pop()
console.log(myArr)
```

Output → [1, 2, 3, 4]

Removes the last element from array

(iii) myArr.unshift(9)
console.log(myArr)

Output→ [9, 1, 2, 3, 4]

This function will add the value in starting of array but this method is not much used because it alter the every index of the array values which is not a good practice.

(iv) myArr.shift()
console.log(myArr)

Output→ [1, 2, 3, 4]

This method removes first index value of array.

(v) console.log(myArr.includes(9))

Output→ False

This method will check if particular value exists in array or not.

(vi) console.log(myArr.indexOf(9))

Output→ -1

This method will return the index of a value in array. If value donot exist it will return -1.

(vi) ~~const~~ const newArr = myArr.join()
console.log(newArr);
console.log(typeof newArr);

Output → 1,2,3,4
String

This will convert the array into string

(vii) const myArr = new Array(1,2,3,4,5,6,7);
const newArr = myArr.slice(0,3);
console.log(myArr);
console.log(newArr);

Output → [1,2,3,4,5,6,7]
[1,2,3]

slice returns a copy of a section of an array. negative index can be used to indicate an offset from the end of the array.
The end index is exclusive.

(viii) const myArr = new Array(1,2,3,4,5,6,7);
const newArr = myArr.splice(1,3);
console.log(myArr);
console.log(newArr);

start index ↓ Total number of elements ↓

Output → [1,5,6,7]
[2,3,4]

splice removes elements from array and return an array containing deleted elements.

Special case

pushing array into array →

```
const marvel_heros = ["thor", "Ironman", "Spiderman"]
const dc_heros = ["superman", "flash", "batman"]

marvel_heros.push(dc_heros)
console.log(marvel_heros)
```

Output → ["thor", "Ironman", "spiderman", ["superman", "flash", "batman"]]

Array take any kind of entry data and in this case it took whole array as an entry data.

→ If you want to concat two arrays i.e combine two arrays in one, we use concat

```
const allheros = marvel_heros.concat(dc_heros)
console.log(allheros)
```

Output → ['thor', 'Ironman', 'spiderman', 'superman', 'flash', 'Batman']

→ There's another way to do the same thing which is known as spreading

```
const all_new_heros = [...marvel_heros, ...dc_heros]
console.log(all_new_heros)
```

Output → ['thor', 'Ironman', 'spiderman', 'superman', 'flash', 'Batman']

suppose you have a messed up array that has multiple arrays inside it and to convert it into simple array we have a special function→

```
const another_array = [1,2,3,[4,5,6],7,[6,7,[4,5]]]
const real_array = another_array.flat(infinity)
```

specify the depth up to which you want to simplify

```
console.log(real_array)
```

Output→ [1,2,3,4,5,6,7,6,7,4,5]

Some more Array methods

(i)
```
console.log(Array.isArray(myArr))
console.log(Array.isArray("vasu"))
```

Output→ True
False

checks if given variable, value or object is array or not

(ii)
```
console.log(Array.from("vasu"))
```

Output→ ['v','a','s','u']

Creates an array from an iterable object.

(iii)
```
let score1 = 100
let score2 = 200
let score3 = 300

console.log(Array.of(score1, score2, score3));
```

Output → [100, 200, 300]

It returns a new array from a set of elements

Object literals

There are two ways to create objects the first one is Object.create in which we get singleton but today we will study about object literals in which there is no singleton.

```
const JsUser = {
        name: "Hitesh",
        age: 18,
        location: "Jaipur",
        email: "hitesh@gmail.com"
    }
```

There are two ways to access the specific value in object →

(i)
```
console.log(JsUser.email)
```

Output → "hitesh@gmail.com"

But this is not a good practice because in some

cases it will not be able to access the value. Especially in the case of symbol and when we pass key as string.

Note → Symbol is data type in Java script

→ To deal with these cases always prefer second method

(ii) console.log (JsUser["email"])

behind the scene the key work as string so add double quotes

now exceptional cases →

```
const JsUser = {
    name: "Hitesh",
    "full name": "Hitesh Choudhary",
    age: 18,
    location: "Jaipur",
    email: "hitesh@gmail.com"
}

console.log(JsUser."full name");
```

→ error

→ So to access full name we use
console.log (JsUser["full name"])

using symbol as a key →

(i) const mySym = Symbol("key1")

```
const JsUser = {
    name: "Hitesh",
    "full name": "Hitesh Choudhary",
    mySym: "mykey1",
    age: 18,
    location: "Jaipur",
    email: "hitesh@gmail.com"
}

console.log(JsUser.mySym)
console.log(typeof JsUser.mySym)
```

Output →

```
mykey1
string
```

now this mySym is of type string but we need it of type Symbol

(ii)

```
const mySym = Symbol("key1")

const JsUser = {
    name: "Hitesh",
    "full name": "Hitesh Choudhary",
    [mySym]: "mykey1",
    age: 18,
    location: "Jaipur",
    email: "hitesh@gmail.com"
}

console.log(JsUser[mySym])
console.log(typeof JsUser[mySym])
```

Output→ mykey1
string

The value will be string in this case too but this is a good practice. But in JsUser key will be [Symbol(Key1)]

Overwriting a value

JsUser.email = "hitesh@chatgpt.com"

If you want that value can't get changed

Object.freeze(JsUser)
JsUser.email = "hitesh@microsoft.com"

now even if you changed any value in JsUser the value will not get changed and yet there will be no shown error.

Passing function as an array key or object key

```
JsUser.greeting = function(){
    console.log("Hello JS user")
}
console.log(JsUser.greeting);
```

Note→ Remember to remove Object.freeze()

Output→ Function (anonymous)

The function didn't get executed but we got the reference of function

```
console.log(JsUser.greeting())
```

Output → Hello JS user
undefined

→ How to access the value from object in function:

```
JsUser.greetingTwo = function(){
    console.log(`Hello JS user, ${this.name}`);
}

console.log(JsUser.greetingTwo());
```

Output → Hello Js user, Hitesh
undefined

why we get undefined will be discussed later

Objects singleton and object defining using constructor

(i)
```
const tinderUser = new Object()
console.log(tinderUser)
```

Output → {}
// This Object will be singleton

(ii)
```
const tinderUser = {}
console.log(tinderUser)
```

Output → {}
// This will be literal not singleton

continuing singleton object →

(i) To add values :-

```
tinderUser.id = "123abc"
tinderUser.name = "Sammy"
tinderUser.isLoggedIn = False

console.log(tinderUser)
```

Output → { id: '123abc', name: 'sammy', isLoggedIn: false }

(ii) nested objects →

```
const regularUser = {
    email: "some@gmail.com",
    fullname: {
        userfullname: {
            firstname: "hitesh",
            lastname: "choudhary"
        }
    }
}

console.log(regularUser.fullname.userfullname);
```

Output → { firstname: 'hitesh', lastname: 'choudhary' }

Combining the Objects

```
const obj1 = { 1: "a" , 2: "b" }
const obj2 = { 3: "a" , 4: "b" }

const obj3 = { obj1, obj2 }

console.log(obj3)
```

Output→ { obj1: { '1': 'a', '2': 'b' }, obj2: { '3': 'a', '4': 'b' } }

// Same as that of array problem on joining

To Solve that Problem→

```
const obj3 = Object.assign(obj1, obj2)
console.log(obj3);
```

Output→ { '1': 'a', '2': 'b', '3': 'a', '4': 'b' }

The Object.assign() static method copies all enumerable own properties from one or more source objects to a target object. It returns the modified target object.

eg→
```
const target = { a: 1, b: 4 };
const source = { b: 2, c: 5 };
const returnedTarget = Object.assign(target, source);
console.log(target)
console.log(returnedTarget)
```

Output → { a:1 , b:2 , c:5 } // target
{ a:1 , b:2 , c:5 } // returnedTarget

now as you can see target got modified but if you don't want target to get modified you can use this syntax →

eg → const returnedTarget = Object.assign ({}, target , source)
console.log (target)
console.log (returnedTarget)

Output → { a:1 , b:4 } // Target
{ a:1 , b:2 , c:5 } // returned Target

→ The other method to do the same is by using spreading as we have done in array:

const obj3 = {...obj1, ...obj2}

Note → when we fetch values from database , we get them in form of array of objects.

```
const users = [
    { },
    { }
]
```

Accessing Keys of Objet

```
const tinderusers = {
        id: '123 abc',
        name: 'Sammy',
        isloggedIn: false
    }
```

```
console.log (Object.keys (tinderusers));
```

Output → ['id', 'name', 'isloggedIn']

Imp

The output we get is in the form of array or it take keys of object and return it in the form of array.

→ we can access the values in the same way and the output again will be in the form of array.

```
console.log (Object.values (tinderUsers));
```

Output → ['123 abc', 'Sammy', false]

→ If we want to access both keys and values we can do that too

```
console.log (Object.entries (tinderusers));
```

Output → [['id', '123 abc'], ['name', 'Sammy'],

['isloggedIn', false]]

→ To check if object has particular property or not:

```
console.log(tinderUsers.hasOwnProperty('isLoggedIn'));
```

Output → True

Destructuring an object

```
const course = {
    coursename: "Js in Hindi",
    price: "999",
    courseInstructor: "hitesh"
}
```

now we have discussed two ways to print element of an object but there is one more way by using de-structure →

```
const {courseInstructor} = course
console.log(courseInstructor)
```

Output → hitesh

→ If you think courseInstructor is a large name and want to give another name, you can do that as well

```
const {courseInstructor: instructor} = course
console.log(instructor)
```

Output → hitesh

The de-structure concept is widely used in React.

JSON API Intro

When we call the API through JSON we get it in object format or array of objects.

eg →
```
{
  "name" : "hitesh",
  "coursename": "JSON in hindi",
  "price": "free"
}
```

It may give error when written in JS code but it is a perfect Syntax

To get the data of API we have some methods and the pass URL in it and the we get the data of that URL in JSON format.
We will learn about them later in detail.

Some time it is hard to understand and work with API data, to understand API format data in detail we have many tools, one of which is jsonformatter.org.

JSON → Java Script object notation

Functions in Java Script

```
function sayMyName () {
    console.log ("Hello");
}
```

↑

// Defining a function

```
sayMyName
```

↑

// This is known as Reference of a function

```
sayMyName()
```

// This is execution of a function

Functions with arguments→

```
function addTwoNumbers (number1, number2) {
    console.log(number1 + number2);
}

addTwoNumbers (3,4)
```

Note→ when we give variable name while defining a function they are known as parameters.
When we call a function to execute and give value they are known as arguments.

you can also store the value from the function in variable and then print it and for that "return" is use. If you don't return a value in a function

and try to store a value in a variable, then print it, the output will be undefined

```
function addTwoNumbers (number1, number2) {
        let result = number1 + number2
        return result
    }

const result = addTwoNumbers (3, 5)
console.log (result)
```

→ you can also print value directly using parameters

```
function loginUserMessage (username) {
    return `${username} just logged in`
}
console.log (loginUserMessage ("hitesh"))
```

↑ if you did not pass anything here in output you will get undefined and to handle it we use if else statements

→

```
function loginUserMessage (username) {
    if (username === undefined) {
        console.log ("Please log in");
    }
    return `${username} just logged in`
}

console.log (loginUserMessage ())
```

Output → please log in
undefined

→ There is another way to check the same in if condition

```
if (!undefined) { }
```

→ you can also provide a default value to parameter

```
function loginUserMessage (username = "sam") {
}
```

Rest operator in functions

This operator is used when we don't know how much parameter we need or how much argument will be passed on function call.
like when we have multiple items in shopping cart and we need to calculate total price.
The rest operator is same spread operator or their notation is same.

```
function calculateCartPrice (...num1) {
    return num1
}
console.log (calculateCartPrice (200, 400, 500))
```

Output → [200, 400, 500]

Passing object and array as arguments

(i)
```
function handleObject (anyobject) {
  console.log(`username is ${anyobject.username}
    and price is ${anyobject.price}`);
}

handleObject ({
      username: "sam",
        price : 399
    })
```

(ii)
```
function returnSecondValue (getArray) {
        return getArray[1]
}
console.log (returnSecondValue([200,400,500,1000]));
```

Scope in Java Script

```
if (true) {
    let a = 10
    const b = 20
    var c = 30
}

console.log(a)
console.log(b)
console.log(c)
```

In the above example, if we try to print a in above example it will give error because the scope of a is limited to if condition or block scope.
Same will be the case with b.
But if we print c there will be no error, it will easily give the output 30 which is not a good thing.
That's the reason why we don't use var keyword.

Anything outside the block scope is known as global variable or scope.
The variable in the global scope can also be used inside the block space but vice versa is not true as discussed above.

```
let a = 300
if (true) {
    let a = 30
    console.log("Inner: ", a)
}
console.log(a)
```

Output → Inner: 30
300

Note → The scope work differently when we are working in inspect console in browser, known as core scope which will be discussed later.

Closure →

It is a vast concept but for now when inner function or object or element can access the variables of outer element.

```
function one (){
        const username = "vasu"

        function two (){
            const website = "youtube
            console.log (username);
        }
        console.log (website); //error because out of
        two ()                          scope
}
one()
```

If we comment the error line, then output will be →

vasu

Same is the case with if else and other blocks. closure will be discussed later.

Important concept

Now there is another way of declaring function using variable and in this case it is known as expression.

```
const addTwo = function (num) {
        return num+2
}
```

now both the ways of defining function are similar but difference occour when we define function later but execute it before defining. In case of expression we get error but in case of normal there will be no error.

eg →

```
addone(5)   // no error
function addone(num){
    return num+1
}

addTwo(5)  // will give error
const addTwo = function(num){
    return num+2
}
```

This concept is known as hoisting (moving declara-tions to the top). It will be discussed in detailed later or you can check from W3 schools.

This and arrow function

this keyword tells us about current context.

```
const user = {
    username: "hitesh",
    price: 999,

    welcomeMessage: function(){
        console.log(`${this.username}, welcome`);
    }
}
```

```
user.welcomeMessage()
user.username = "sam"
user.welcomeMessage()
```

Output → hitesh , welcome
sam , welcome

Note → ` ` ` These are known as bactics

Now what if we just console log this →

As we know that this refer to current context if we are in node enviorment then console.log will give empty object.
and if it is in a scope it will refer to object in it.

```
const user = {
    username: "hitesh",
    price: 999,

    welcomeMessage: function () {
        console.log (`${this.username}, welcome`);
        console.log (this);
    }
}

user.welcomeMessage()
console.log(this)
```

Output → hitesh, welcome
{
username: 'hitesh',
price: 999,
welcomeMessage: [Function: welcomeMessage]
}
{}

This will be output when you are in window enviorment i.e this will give empty object but when in browser and then you console.log(this) in inspect console then output will be window object.

now what if we just console.log(this) in function

(i) function chai() {
console.log(this)
}
In output you will get many different values rather than empty object.

(ii) function chai() {
let username = "hitesh"
console.log(this.username);
}
chai()

Output → undefined

Same will be the output if you declare function using variable.

Arrow function

```
const chai = () => {
    let username = "hitesh"
    console.log(this);
}
chai()
```

Output → {}

In arrow function we get empty object unlike normal function. But this is not only difference and also if we print this.username we will still get undefined.

with parameters

```
const addTwo = (num1, num2)=>{
        return num1+num2
}
```

now when we know that you have just single value return like in above example we can do it using implicit return without using return keyword.

```
const addTwo = (num1, num2) => num1+num2
```

OR

```
const addTwo =(num1, num2) => (num1+num2)
```

This technique is widely used in react.

This method is used when we have to return objects

Immediately Invoked function Expression (IIFE)

now there are two needs of IIFE the first is when we immediately and secondly to prevent the function from global scope pollution. function has access to all the global variables.

The IIFE is done when we define and call the function at same time.

()() → call it immediately
↓
define
function

```
(function chai(){
    console.log(`DB CONNECTED`)
})()
```

Output → DB CONNECTED

Now there will be one error when you again define a function or any code block after IIFE, because IIFE do not know where to stop the flow of code so you always need to explicitly stop the IIFE by adding ;

```
(function chai{
    console.log(`DB CONNECTED`)
})();
```

This is named IIFE

```
(() => {
    console.log(`DB CONNECTED TWO`);
})();
```

Output → DB CONNECTED
DB CONNECTED TWO

you can also use arrow function as IIFE

IIFE with parameters

```
((name) => {
    console.log(`DB CONNECTED TWO ${name}`);
})('vasu');
```