|| Hare Krishna ||

Miracle
Pg.No :
Date: / /20

ES6+

→ ES6+ stands for ECMA Script 6.

→ Was created to standardize Javascript. Newer version of Javascript.

✱ Template literals :

` `

e.g. let word1 = "Khushi"
let word2 = "Patel"

Normal : const fullName = word1 + word2

` ` : const fullName = `${word1} ${word2}`

→ Allows writing in multiple lines.

e.g. `${word1}
${word2}`

console.log(fullName) → Not in console.log but in HTML (innerText) or in console of browser

→ Khushi
Patel

✱ ~~Destructing~~ Objects
Destructuring

e.g. const player = {
name: 'James',
club: 'LA',
address: {

```
        city: 'Los Angeles'
    }
};
```

New:
```
const { name, club, address: { city } } = player

console.log(`${name} ${city} ${club}`)
```

Old:
```
console.log(player.name)
    "    "(player.address.city)
```

* Destructuring arrays :

e.g.
```
let name = ['Khushi', 'Jamus']

let [firstName, lastName] = ['Khushi', 'Jamus']

console.log(lastName)
```
→ Jamus

```
lastName = 'Patel'

console.log(lastName)
```
→ Patel

* Object Literals :

e.g.
```
function address(city, state) {
    const newAdd = { city, state }
    console.log(newAdd)
}
```

address('Austin', 'Texas')

→ { city : "Austin", state : "Texas" }

automatically (no need to write {city: city, state: state} in function)

* For loop

for(const income of incomes){

}

* Spread Operator

e.g. let contacts = [" ", " ", " "]

let personalFrnds = contacts

Now, if you add or delete anything in contacts, the same would happen in personalFrnds.

But you don't want that. You want original contacts array only. To do this, use spread operator.

let personalFrnds = [... contacts]

spread

You can also add,

let peFr... = [" ", ...contact, " "]

→ For objects :

```
let person = {
    name : " ",
    age : 25,
    city : "Poland"
}
```

If you want a copy of this,

```
let employee = {
    ...person
    //can also add new values
    salary :

}
```

✱ Rest Operator :

→ When you have a function and don't know the number of arguments it would have, we use rest operator.

e.g.

```
function add(...nums){
        console.log(nums)
}

add(4,5,6,7,8,12)
```

→ [4,5,6,7,8,12] → Array in output (iterable)
↳ Advantage

→ Not iterable if you want:

```
function add(nums) {
        console.log(arguments)
}

add(4,5,6,12)
```

⟹ {0:4, 1:5, 2:6, 3:12} → object as output

✱ Arrow Functions

```
// function declaration
function breakfastMenu() {
    ....
}

// anonymous declaration (No name func)
const Menu = function() {

}
```

Arrow:

```
const dinnerMenu = () => {
                    ↑ parameter if it is there
    :
}
```

If one line inside func, can remove { }

" " parameter, can remove ()

e.g.

```
const dinnerMenu = food => `I love ${food}`

console.log(dinnerMenu("burger"))
```

⇒ I love burger

* Default Params (parameter)

→ When you don't pass any argument but defined a parameter in the func, we can use default params.

e.g.

```
const leadSinger = (artist="someone") => {
        console.log(`${artist}`)
}

leadSinger()
leadSinger("Khushi")
```

⇒ someone
⇒ Khushi

* includes() → Returns boolean

e.g.

```
let numArray = [1,2,3,4,5]

console.log(numArray.includes(2))
⇒ true
                "          "(6))
                ⇒ false
```

✱ Let and const

→ const = Constant, cannot change its value.

We can modify [] arrays and {} objects e.g. .push(), ___.name = "Khushi" in const but we cannot change primitive data types like strings, numbers, booleans etc.

→ let = Can be changed later on.

e.g.

```
let num = 5
num = 7
console.log(num)
⇒ 7
```

```
const = 5 num = 5
num = 7
console.log(num)
⇒ Error (cannot assign to a const)
```

✱ Import and export

→ Can import and export data from one file to another in a project / folder

- example.js

```
export const data = [1, 2, 3]
```

- index.js

```
import {data} from './example.js'

console.log(data)
⇒ [1, 2, 3]
```

Miracle
Pg.No :
Date: / /20

padStart() and padEnd()

→ New javascript methods
→ Allow formatting a string by adding padding characters at the start or end.

e.g let example = 'Khushi'

console.log(example.padStart(10, 'a'))
⇒ aaaaKhushi
10

console.log(example.padEnd(10, 'a'))
⇒ Khushiaaaa
10

console.log(example.padStart/End(6, 'a'))
⇒ Khushi
Because only 6 letters allowed and its already occupied by Khushi

console.log(example.padStart(10))
⇒ Khushi
added empty string

" ", " ".padStart(10).length))
⇒ 10

console.log(example.padStart/End(1))
⇒ Khushi
doesn't make sense, will return same result as original

" ", " "(1).length)
⇒ 6

Miracle
Pg.No :
Date: / /20

Classes

animal.js

→
```
export class Animal {

    constructor(type, legs) {
        this.type = type
        this.legs = legs
    }

    makeNoise(sound = 'Loud Noise') {
        console.log(sound)
    }

    get metaData {
        return `Type: ${this.type}, Legs: ${this.legs}`
    }

    static returnIO() {
        return 10
    }
}

export class Cat extends Animal {
    constructor(type, legs, tail) {
        super(type, legs)
        this.tail = tail
    }
}
```

object name

input

constructor

function

get method

Static methods can be called on class itself. Cannot call on an object but directly on class. e.g. Animal.returnIO()

static method

and constructor no koi use nai hoto, kind of separated from class

child class

parent class

get the method constructor of parent class using super keyword

// We can also override parent class function in child class

e.g export class Cat extends Animal {

```
    makeNoise(sound = "Meow"){
        console.log(sound)
    }
}
```

index.js

```
import { Animal, Cat } from './animal.js'

let cat = new Animal('cat', 4)
```

cat → object; Animal → of class Animal ~~type, leg~~

```
console.log(cat)
```

⇒ Animal { type: "Cat", legs: 4 }

```
console.log(cat.legs)
```

⇒ 4

```
cat.makeNoise()
```

⇒ Loud Noise

```
cat.makeNoise("meow")
```

⇒ meow

} ~~function~~

```
console.log(cat.metaData)
```

⇒ Type: Cat, Legs: 4

Miracle
Pg.No :
Date: / /20

console.log(Animal.returnIO())
⇒ 10

directly on class name

Static method

Cannot be called by object cat

// child class

let cat = New Cat('Cat', 4)

cat.MakeNoise()
⇒ meow

→ can use parent class method in child class

console.log(cat.mataData)
⇒ Type: Cat, legs: 4

✱ Promises:

→ The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

→ Returns single value based on the operation being rejected or resolved.

→ 3 stages:

Pending: Initial stage of each promise. Represents that result has not been computed yet.

Fulfilled: Operation has been completed.

Miracle
Pg.No :
Date: / /20

Rejected: Failure occured during computation

→ Used for: To fetch data from an API or in some asynchronous functions that you are trying to run

→ Syntax:

```
const promise = new Promise((resolve, rejected) => {
                                    :
                                    :
                                })
```

e.g.

```
const buyFlightTicket = () => {

    return new Promise((resolve, reject) => {

        setTimeout(() => {
            const error = false

            if(error) {
                console.log("Sorry")   reject("Sorry...")
            } else {                   reject
                console.log("Successful")
            }                          resolve
        }, 3000)                       resolve("Si...")
    })   timer (3 seconds)

}
```

Miracle
Pg.No :
Date: / /20

buyFlightTicket().
.then((success) => console.log(success))
fulfilled (resolve)
.catch((error) => console.log(error))
error (reject)

// .then() → Takes two parameters. First one is invoked when Promise is fulfilled, the second one (optional) gets invoked when Promise is rejected.

This method gets invoked when the Promise is either fulfilled or if rejected.

.then(success, error)
when fulfilled (optional) when rejected

// .catch() → Handles failures and rejections. Takes only one functional argument to handle errors.

.catch(error)

Output:

⇒ Successful

If const error = true

⇒ Sorry

Miracle
Pg.No :
Date: / /20

✱ Fetch()

→ The fetch API provides a Javascript interface for accessing and manipulating parts of the protocol, such as requests and responses.

→ Provides a global fetch() method.

e.g.
```
fetch('https:.....')
    .then((response) => response.json())
    .then((data) => console.log(data))
```

POST method:

```
fetch('.....', {

    method: 'POST',
    body: JSON.stringify({
        name: " "
        email: " "
        body: " "
        :
        :
    })
})
.then((response) => response.json())
.then((data) => console.log(data))
```

Miracle
Pg.No :
Date: / /20

Async & Await

→ JavaScript is synchronous

→ Lots of functionalities that make our code asynchronous. One of them is Async/Await functionality.

→ Async: Allows us to write promise based code as if it was synchronous.
→ Will always return a value.

→ Await: Used to wait for the promise.
→ Used with async block only.
→ Waits until the promise returns a result.

e.g.
```
const photos = []

async function uploadPhoto() {

    let uploadStatus = new Promise( (resolve, reject) => {
        setTimeout( () => {
            photos.push("ProfilePic")
            resolve("Photo uploaded")
        }, 3000)
    })
let result = await uploadStatus
console.log(result)
console.log(photos.length)
}
```

⇒ Without async and await
Promise
0
Because JS is synchronous. It won't wait for 3 seconds to run promise. It will move ahead

⇒ With async and await
Photo Uploaded
1

* Sets :
same as python (sees unique value (no duplicates))

```
const exampleSet = new Set([1,1,1,2,2,2])

exampleSet.add(5)

console.log(exampleSet.size)
⇒ 3  // 1,2,5

exampleSet.add(17).add(20)...

exampleSet.delete(5)

console.log(exampleSet.size)
⇒ 4  // 1,2,17,20

console.log(exampleSet.delete(2))
⇒ true // this will return true
          or false. If value present
          in set, then true else false

console.log(exampleSet.has(2))
                    ↑
              true or false return
```