Intro – ES6

Topics:

- ES6
- Let & const

History:

Around 2008 – 2010, the time when internet started to boom. At that time, different developers coming in web development areathere were finding out many issues that were more or less very complicated to actually change from someone who has already studied Java, C++ and coing back to scripting. The environment and the way of execution was getting very different. Now all of this cannot be accomplished in a day. It is a long and tedious process. Some team starts preparing a feature which has to go through multiple review process in the committee of W3C. Afther this, the specification is gven to different browser vendors.

After all this, to put the new features into JavaScript, they have to wait for an engine. Like edge was previously using chakra engine, but now it is chromium. So the thing is when ECMA script comes, first it gets updated in JavaScript engine. Chromium uses V8 Engine which is another decore acutal engine on which Javascript actually runs on. So V8 is updated then chromium is updated and then the web browsers are updated.

ES5 – 2005
ES6 – 2016

When ES6 was released, it had resolved quite a lot of issues and gave many new features to build something more fluidly. W3C communite decided instead of giving bulk update after 5-6years, they started giving updates every year.

ES2017 – 2017
ES2018 – June 2018
ES2019 – June 2019
ES2020 – June 2020

We have lot of features specific to ES6. We will be knowing about babel, which allows us to write any new feature of Javascript and babel will convert it inot a previous version so that the code does not fail.

let & const:

Initially, we were writing

```
var name = 'Yash'
```

The issue with **var** is

- Scoping
- Resigning values
- Hoisting

let & **const** are also a way of defining variables.

```
console.log(name)
let name = 'Yash'
```

This would give an error, "**Uncaught ReferenceError: Cannot access 'name' before initialization**"

If we replace **let** with **var**, we get nothing, as it is hoisted over and it has no value. But when we put let instead of var, we get an error.

Understanding:

Let and **const** does not get hoisted. We have to define or initialize variables (accordingly) upfront before actually using it. Now if we change the code to

```
let name = 'Yash'
console.log(name)
```

this will run. Output: Yash

```
let name = 'Yash'
console.log(name)

name = 'Goku'
console.log(name)
```

Output:

Yash
Goku

When you define **let** it is understood that this value can change.

If we use **`const`** instead of **`let`**,

```
const name = 'Yash'
console.log(name)

name = 'Goku'
console.log(name)
```

Output:

Uncaught TypeError: Assignment to constant variable

This means that when you define a variable **`const`**, you cannot reassign it or cannot define another value to it. **`const`** is basically for defining constants.

We can resign the const if the value is not a primitive value. Primitive values are integers, strings and Booleans.

So, if we declare an array as

```
const arr = ['Yash', 'Goku']
console.log(arr)

arr[0] = 'ES'
console.log(arr)
```

The moment you use a complex or non-primitive data structure, we notice that we can change the value or variable from **`'Yash'`** to **`'ES6'`**, even though we declared the **`arr`** as **`const`**. if we use **`let arr = 123`**, will throw an error, as any variable that is declared by **`const`**, cannot be reassigned. We can only change the content but we won't be able to redeclare the variable.

If a variable is defined inside a function, then it is available to all the children function.

If we want the same variable and the value should be different not relevant to the previous declaration. It might give us unexpected variable or hoisting issues. In order to solve this, we use **`let`** and **`const`**. **`let`** and **`const`** are block scope. If we have defined one variable in one block with a value, you can declare the same variable with another value in another function.

Ex:01

```
function x() {
  let abc = 123
  console.log(abc)
  {
    let abc = 'Yash'
    console.log(abc)
  }
}
```

In this it is predictable that the value of **abc** would be 123 and **'**Yash**'** accordingly. If we declare another **const abc** inside the second block, and give a different value, it would again change its value.

NOTE: Always try to use **let** and **const**.

Template Literals:

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

Ex: 02

```
let firstName = 'Yash'
let lastName = 'Sinha'

const fullName = firstName + " " + lastName
console.log(fullName)
```

But if have a use case where we have to concatenate lot of things,

```
let firstName = "Yash's" + " is talking " + "class"
let lastName = 'Sinha'

const fullName = firstName + " " + lastName
console.log(fullName)
```

It would achieve the same thing, but there is another cleaner way, which is template literal.

```
let firstName = "Yash"
let lastName = 'Sinha'

const fullName = `The name of the person is ${firstName} ${lastName}`
```

```
console.log(fullName)
```

this is one uses of template literals. Another use is we can put in expression.

```
let firstNumber = 100
let lastNumber = 200

const sumNum = `The sum of two numbers is ${firstNumber + lastNumber}`
console.log(sumNum)
```

Output:

The sum of two numbers is 300.

New String Methods

startsWith()

The startsWith() method determines whether a string begins with the characters of a specified string, returning true or false as appropriate. true if the given characters are found at the beginning of the string; otherwise, false.

This method lets you determine whether or not a string begins with another string. This method is case-sensitive.

```
const str = "This is a very long sentence."

console.log(str.startsWith('this'))
```

---

endsWith()

The endsWith() method determines whether a string ends with the characters of a specified string, returning true or false as appropriate. true if the given characters are found at the end of the string; otherwise, false.

```
const str = "This is a very long sentence."

console.log(str.endsWith('long')) // false
```

---

includes()

The includes() method performs a case-sensitive search to determine whether one string may be found within another string, returning true or false as appropriate.

```
const str = "This is a very long sentence."

console.log(str.includes('very')) // true
```

---

repeat()

The repeat() method constructs and returns a new string which contains the specified number of copies of the string on which it was called, concatenated together.

A new string containing the specified number of copies of the given string.

```
const str = "This is a very long sentence."

console.log(str.repeat(2))
```

Arrow Functions:

An arrow function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

Differences & Limitations:

- Does not have its own bindings to **`this`** or **`super`**, and should not be used as **`methods`**.
- Does not have **`arguments`**, or **`new`**.**`target`** keywords.
- Not suitable for **`call`**, **`apply`** and **`bind`** methods, which generally rely on establishing a **`scope`**.
- Cannot be used as **`constructors`**.
- Cannot use **`yield`**, within its body.

Ex:03

```
function x() {

}

const add = function(num1, num2) {

  return num1 + num2

}

console.log(add(10, 5))
```

This is how we write a function, but using arrow function, instead of writing the text function, we can replace it by **`'=>'`**. This is shortening the code. The second thing is when remove the curly braces as well. The arrow function is always used either as callback function or in the function expression.

If we have a one liner expression, we can get rid of the curly braces as well. So, we can write it as;

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

Ex: 04

```
const square = (num1) => num1**2
console.log(square(10))
```

Since we are only using one variable, we can write it as

```
const square = num1 => num1**2
console.log(square(10))
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Ex:05

```
const arr = [5,10,15]
cont squaredArr = arr.map(item => item**2)

console.log(squaredArr)
```

This is another benefit of shortening the code as we always have to write short code.

De-Structuring:

Destructuring assignment allows you to assign the properties of an array or object to variables using syntax that looks similar to array or object literals. This syntax can be extremely terse, while still exhibiting more clarity than the traditional property access

```
const person = {
  firstName: 'Yash',
  lastName: 'Sinha',
  age: 80,
  height: 200,
  profession: 'unknown'
  sayHi: () => {}
}

function greeter(personObj) {

  console.log(`Hello, I am ${personObj.firstName}
${personObj.lastName}`)
}
```

```
greeter(person)
```

We can also write it as

```
function greeter(personObj) {

  const fName = personObj.firstName
  const lName = personObj.lastName
  const {firstName, lastName} = personObj
  console.log(`Hello, I am ${fNametoUpperCase} ${lName.toUpperCase}`)
}
greeter(person)
```

We can iterate on the keys.

```
function greeter(personObj) {
     const {firstName, lastName} = personObj
     console.log(`Hello, I am ${fNametoUpperCase} ${lName.toUpperCase}`)
}
greeter(person)
```

We created same as property name, which enhances readability and shortens the code. This is one example of object de-structuring.

For an array element instead of writing something like,

```
const arr = [10, 20, 30]

let firstItem = arr[0]
let secondItem = arr[1]
```

we can write it as,

```
const arr = [10, 20, 30]
let [firstItem, secondItem, thirdItem] = arr

console.log(firstItem)
console.log(secondItem)
```

Output:

10
20
30

In array we can define variable names and in objects we can define property names.

https://hacks.mozilla.org/2015/05/es6-in-depth-destructuring/