Save it

Like it

Quick
Es6 Guide

@coder_aishya

Let & const Keywords

- Variables were previously declared using **"var" which had function scope** and were hoisted to the top within its scope. It means that **a variable can be used before declaration.**
- But, the **"let" variables and constants have block scope** which is surrounded by curly-braces "{}" , they are **not hoisted** & **cannot be used before declaration.**
- The **new const keyword makes it possible to define constants.** Constants are read-only, you **cannot reassign new values to them.**

@coder_aishya

Let example:

```
let  x = 10; // Here x is 10

{
  let x = 2; // Here x is 2
}


document.getElementById("demo").innerHTML = x;
//Output : 10
```

const example:

```
var x = 10;// Here x is 10
{
  const x = 2;// Here x is 2
}

document.getElementById("demo").innerHTML = x;
//Output : 10
```

@coder_aishya

Arrow Functions

- It provides a more concise syntax for writing function expressions by removing the "function" and "return" keywords.
- Arrow functions are defined using the fat arrow (=>) notation.
- Unlike ordinary functions, arrow functions do not have their own this keyword.
- The value of this inside an arrow function is always bound to the value of this in the closest non-arrow function.
- **Arrow functions are not hoisted.** They must be defined before they are used.

@coder_aishya

Arrow function Example :

```
// ES5 Function Expression
var sum = function(a, b) {
    return a + b;
}
console.log(sum(2, 3)); // Output : 5

// ES6 Arrow function
var sum = (a, b) => a + b;
console.log(sum(2, 3)); // Output : 5
```

@coder_aishya

Multi-line Strings

Users can create multi-line strings by using back-ticks(`). In ES5 we needed to use '\n' for multi line statements

```
//ES5 Sytax
var poemData = 'Johny Johny Yes Papa,\n'
             + 'Eating sugar?  No, papa!\n'
             + 'Telling lies? No, papa!\n'
             + 'Open your mouth Ah, ah, ah!'

//ES6 syntax
let poemData = `Johny Johny Yes Papa,
                Eating sugar?  No, papa!
                Telling lies? No, papa!
                Open your mouth Ah, ah, ah!`
```

@coder_aishya

Template Literals

ES6 introduces very simple string templates along with **placeholders for the variables**. The syntax for using the string template is **${PARAMETER}** and is used inside of the back-ticked string.

```
//ES5 Sytax
var name = 'Your name is ' + firstName + ' ' + lastName + '.'

//ES6 syntax
var name = `Your name is ${firstName} ${lastName}.`
```

@coder_aishya

Default Parameters

ES6 allows **function parameters to have default values**.But, in ES5, OR operator had to be used.

```
//ES5 syntax
var calculateArea = function(height, width) {
    height =  height || 50;
    width = width || 80;
    // logic
}

//ES6 syntax
let calculateArea = function(height = 100, width = 50) {
    // logic
}
```

@coder_aishya

Destructuring Assignment

- The destructuring assignment is an expression that makes it easy to extract values from arrays, or properties from objects, into distinct variables.
- There are two types of destructuring assignment expressions, namely, Array Destructuring and Object Destructuring.

```
//Array Destructuring
let fruits = ["Apple", "Banana"];
let [a, b] = fruits; // Array destructuring assignment
console.log(a, b);
//OUTPUT: Apple Banana

//Object Destructuring
let person = {name: "Peter", age: 28};
let {name, age} = person; // Object destructuring assignment
console.log(name, age);
//OUTPUT: Peter 28
```

@coder_aishya

Enhanced Object Literals

ES6 provides enhanced object literals which make it easy to quickly create objects with properties inside the curly braces.

```
function getLaptop(make, model, year) {
    return {
        make,
        model,
        year
    }
}

getLaptop("Apple", "MacBook", "2015");
```

@coder_aishya

Promises

Promises are used for asynchronous execution. We can use promise with the arrow function

```
var asyncCall =  new Promise((resolve, reject) => {
    // do something
    resolve();
}).then(()=> {
    console.log('DONE!');
})
```

I have already made a detailed post on Promises. Do check that out on my page

@coder_aishya

Classes

- Classes are introduced in ES6 which looks similar to classes in other object-oriented languages, such as C++, Java, PHP, etc. But, they do not work exactly the same way.
- We can create class in ES6 using **“class” keyword.**
- ES6 classes make it simpler to create objects, implement inheritance by using the **"extends" keyword** and also reuse the code efficiently.

@coder_aishya

Class Example

```
class UserProfile {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getName() {
      console.log(`The Full-Name is ${this.firstName}
                   ${this.lastName}`);
    }
}
let obj = new UserProfile('John', 'Smith');
obj.getName();
// output: The Full-Name is John Smith
```

@coder_aishya

Function Rest Parameter

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

@coder_aishya

Modules

- ES6 introduced a new feature called modules, in which each module is represented by a separate ".js" file.
- We can use the "import" or "export" statement in a module to import or export variables, functions, classes or any other component from/to different files and modules.

@coder_aishya

Modules Example

```
export var num = 50;
export function getName(fullName) {
    //logic
};
```

```
import {num, getName} from 'module';
console.log(num); // 50
```

@coder_aishya

Map

- Before ES6, JavaScript had only arrays & Objects as Data structures
- Objects are used for storing keyed collections.
- Arrays are used for storing ordered collections.

ES6 Data Structures

- Map
- Set
- WeakMap
- WeakSet

@coder_aishya

Map Overview

- Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows **keys of any type.**
- A Map remembers the original insertion order of the keys.
- A Map has a property that represents the size of the map.

@coder_aishya

Methods and properties

- **new Map()** – creates the map.
- **map.set(key, value)** – stores the value by the key.
- **map.get(key)** – returns the value by the key, undefined if key doesn't exist in map.
- **map.has(key)** – returns true if the key exists, false otherwise.
- **map.delete(key)** – removes the value by the key.
- **map.clear()** – removes everything from the map.
- **map.size** – returns the current element count.

@coder_aishya

Map creation

You can create a JavaScript Map by:

- Passing an Array to new Map()
- Create a Map and use Map.set()

new Map()

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

@coder_aishya

Map.set()

```
// Create a Map
const fruits = new Map();

// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);
```

The set() method can also be used to change existing Map values.

@coder_aishya

Map.get()

gets the value of a key in a Map

```
fruits.get("apples");    // Returns 500
```

Create a map from an object

```
let srcObject = {
  name: "John Snow",
  title: "King in the North"
};
let map = new Map(Object.entries(srcObject));

console.log( map.get('name') ); //output:  John Snow
```

@coder_aishya

Map Iteration

For looping over a map, there are 3 methods:

- **map.keys()** – returns an iterable for keys,
- **map.values()** – returns an iterable for values,
- **map.entries()** – returns an iterable for entries [key, value], it's used by default in for..of.

@coder_aishya

Map Iteration

For looping over a map, there are 3 methods:

- **map.keys()** – returns an iterable for keys,
- **map.values()** – returns an iterable for values,
- **map.entries()** – returns an iterable for entries [key, value], it's used by default in for..of.

@coder_aishya

Set

- A Set is a special type collection – ordered “list of values” (without keys), where each value may occur only once
- The Set is well optimized in performance to create a unique arrays
- One difference between ES6 Sets and those in other languages is that the order matters in ES6

@coder_aishya

Methods

- **new Set(iterable)** : creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.
- **set.add(value)** : adds a value, returns the set itself.
- **set.delete(value)** : removes the value, returns true if value existed at the moment of the call, otherwise false.
- **set.has(value)** : returns true if the value exists in the set, otherwise false.
- **set.clear()** : removes everything from the set.
- **set.size** : is the elements count.

@coder_aishya

Set creation

You can create a JavaScript Set by Passing an Array to new Set()

The new Set() Method

Pass an Array to the new Set() constructor:

```
const letters = new Set(["a","b","c"]);
```

@coder_aishya

Adding elements to a Set

```
let vegetables = new Set( );

let cucumber = { name: "cucumber" };
let onion = { name: "onion" };
let potato = { name: "Potato" };

// add multiple times
vegetables.add(cucumber);
vegetables.add(cucumber);
vegetables.add(onion);
vegetables.add(potato);
vegetables.add(onion);
vegetables.add(potato);

// set is a unique values collectionn
console.log( vegetables.size ); //output: 3
```

@coder_aishya

Set Iteration

The same methods Map has for iterators are also supported in Set:

- **set.keys()** – A Set has no keys. keys() returns the same as values().This makes Sets compatible with Maps
- **set.values()** – returns an Iterator object containing all the values in a Set
- **set.entries()** – A Set has no keys. entries() returns [value,value] pairs instead of [key,value] pairs.This makes Sets compatible with Maps

@coder_aishya

Set Iteration Example

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Elements
let text = "";
letters.forEach (function(value) {
  text += value + "<br>";
})

document.getElementById("demo").innerHTML = text;

//Output:
a
b
c
```

@coder_aishya

When to use What?

- Use a Set when your dataset needs to be composed of unique values
- Use a Map when you have pairs of associated data. You map the keys to the values

@coder_aishya

What is Garbage Collection ?

JavaScript Garbage Collection is a form of memory management whereby objects that are no longer referenced are automatically deleted and their resources are reclaimed.

Weak Collections

- Map and Set's references to objects are strongly held and will not allow for garbage collection.
- WeakMap and WeakSet ES6 collections are 'weak' because they allow for objects which are no longer needed to be cleared from memory.

@coder_aishya

WeakMap

- A WeakMap is a collection of key/value pairs whose **keys must be objects only**. Primitive data types as keys are not allowed
- WeakMap does not support iteration and methods keys(), values(), entries(), so there's no way to get all keys or values from it.

Methods

- weakMap.get(key)
- weakMap.set(key, value)
- weakMap.delete(key)
- weakMap.has(key)

@coder_aishya

WeakMap Example

```
const aboutAuthor = new WeakMap( ); // Create New WeakMap
const currentAge = {}; // key must be an object
const currentCity = {}; // keys must be an object

aboutAuthor.set(currentAge, 30); // Set Key Values
aboutAuthor.set(currentCity, 'Denver'); // Key Values can
be of different data types

console.log(aboutAuthor.has(currentCity)); // Test if
WeakMap has a key

aboutAuthor.delete(currentAge); // Delete a key
```

@coder_aishya

UseCases of WeakMap

To keep an object's private data private

```
var Person = (function() {
  var privateData = new WeakMap();

  function Person(name) {
    privateData.set(this, { name: name });
  }

  Person.prototype.getName = function() {
    return privateData.get(this).name;
  };

  return Person;
}());
```

@coder_aishya

To keep track of DOM node edits, removals, and changes

```
_makeClone() {
  this._containerClone = this.container.cloneNode(true);
  this._cloneToNodes = new WeakMap();
  this._nodesToClones = new WeakMap();

  ...

  let n = this.container;
  let c = this._containerClone;

  // find the currentNode's clone
  while (n !== null) {
    if (n === this.currentNode) {
    this._currentNodeClone = c;
    }
    this._cloneToNodes.set(c, n);
    this._nodesToClones.set(n, c);

    n = iterator.nextNode();
    c = cloneIterator.nextNode();
  }
}
```

@coder_aishya

Caching

cache.js

```
let cache = new WeakMap();

// calculate and remember the result
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculate the result for */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}
```

main.js

```
let obj = {/* some object */};

let result1 = process(obj);
let result2 = process(obj);

// ...later, when the object is not needed any more:
obj = null;

// Can't get cache.size, as it's a WeakMap,
// but it's 0 or soon be 0
// When obj gets garbage collected, cached data will be
removed as well
```

@coder_aishya

WeakSet

- WeakSet behaves similarly to WeakMap
- Similar to Set, but we can only add objects (not primitive types).
- *An object exists in the set as long as it can be accessed from elsewhere.
- *Like set, it supports add, has, and delete, but not size, keys(), and iterations.

Methods

- weakSet.add(key)
- weakSet.delete(key)
- weakSet.has(key)

@coder_aishya

Example : we are on a page where we are showing Messages and we are showing unread messages as notifications. When a message is deleted, it will automatically be deleted from unread messages.

```
let messages = [
    {text: "Merhaba", from: "Oğuz"},
    {text: "Naber?", from: "Sezer"},
    {text: "Dudu Dudu", from: "Tarkan"}
];
let read = new WeakSet();

read.add(messages[0]);
read.add(messages[1]);
read.add(messages[0]);
read.add(messages[0]);
//A message can be read more than once. But the array will
not change

messages.shift();
//When the message is deleted, it is also deleted from the
read.
```

@coder_aishya

In ES6, The three dots operator (...) means two things:

- **The spread operator**
- **The rest operator**

Spread Operator

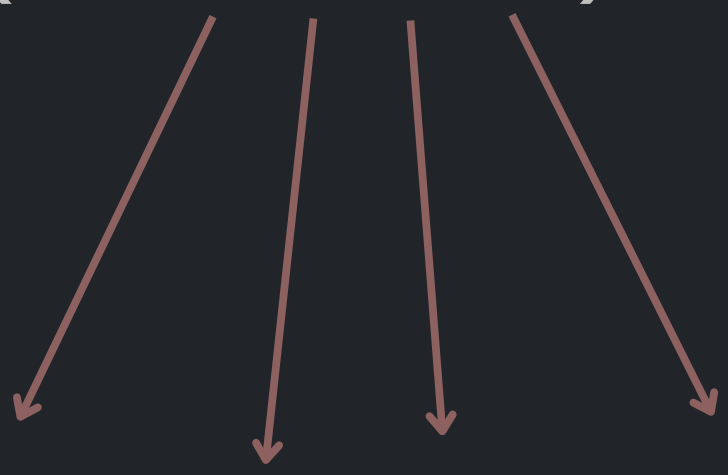
- When used as a spread operator, the three dots operator spreads the elements of an array where zero or more arguments are expected.
- When dealing with objects, you can use the spread operator to spread key-value pairs where expected.

@coder_aishya

Spread Syntax

let numbers = [3,6,2,4]

operate(...numbers)



operate(3,6,2,4)

Spread Syntax in function calls

```
someFunc(...iterObj);
// pass all the elements of iterObj
as arguments to someFunc
```

@coder_aishya

Spread Syntax with array literals

```
["el1", "el2", "el3", ...arr];
// add the elements of arr to the array
```

Spread Syntax with object literals

```
let objCopy = { ...obj };
// Spread all key-value pairs from obj
to a new object
```

@coder_aishya

Examples

with a Function

```
function showCoords(x, y, z) {
    console.log(`x: ${x}, y: ${y}, z: ${z}`);
}
const coords = [2, 1.5, 3.5];
showCoords(...coords); // Prints "x: 2, y: 1.5, z: 3.5"
```

with an Array Literal

```
const boys = ['Bob', 'Charlie'];
const girls = ['Alice', 'Diana'];
const all = [...boys, ...girls];
console.log(all);
// ["Bob", "Charlie", "Alice", "Diana"]
```

@coder_aishya

```js
const boys = ['Bob', 'Charlie'];
const girls = ['Alice', 'Diana'];

const all = ["Eric", ...boys, ...girls, "Gabriel"];

console.log(all);
// ["Eric", "Bob", "Charlie", "Alice", "Diana", "Gabriel"]
```

with Object Literal

```js
let obj1 = { test: 'value', x: 10 };
let obj2 = { test: 'other value', y: 20 };
let combined = {...obj1, ...obj2};
console.log(combined);
// Object { test: "other value", x: 10, y: 20 }
```

@coder_aishya

Rest Operator

- Spread syntax “expands” an array into its elements, while rest syntax collects multiple elements and “condenses” them into a single element
- The rest syntax makes it possible to create a function that accepts any number of arguments.
- It is used for destructuring arrays and objects.

@coder_aishya

Rest Syntax

```
function f(a, b, ...moreArgs) {
    // actions
}
```

Here you can input any number of arguments after the first and the second one.

Restrictions with the Rest Parameter

- There can be only one rest parameter.
- The rest parameter has to be the last parameter in the function.

@coder_aishya

Example

if we want a function to add a list of numbers, that had no definite size of the list,this would work

```
function adder (...numbers) {

    let total = 0

    // You can iterate through the REST operator just like any regular array, or [args] you might pass.
    numbers.forEach((n, i) => {
        // Type check to avoid turning this thing into a string, or creating some other error.
        if (typeof n === 'number') {
            total += n
        } else {
            console.log('can\'t add item at index' + i + '.')
        }
    })

    return total
}

adder(1, 2, 3, 4, 5, 6, 7, 8);
```

@coder_aishya

In the example below, de-structuring of elements happened for the first two elements and the leftover elements are collected into a new sub-array.

```
//Simple Array
const studentsRollNoArr=[1,2,3,4,5];

//Using Rest operators for Destructing
const [first,second,...others]=studentsRollNoArr;

console.log("First element of Array",first);
//Output:1

console.log("Others of Array",others);
//Output:[3,4,5]
```

@coder_aishya

Follow me for more

aishwarya-dhuri

coder_aishya