|  |
| --- |
|  |
|  | JavaScript ES6 brings new syntax and new awesome features to make your code more modern and more readable. It allows you to write less code and do more. |
|  | ES6 introduces us to many great features like arrow functions, template strings, class destruction, Modules… and more. |
|  |  |
|  | **Q)list the top 10 best ES6 features** |
|  |  |
|  |  |
|  | 1.Default Parameters |
|  | 2.Template Literals |
|  | 3.Multi-line Strings |
|  | 4.Destructuring Assignment |
|  | 5.Enhanced Object Literals |
|  | 6.Arrow Functions |
|  | 7.Promises |
|  | 8.Block-Scoped Constructs Let and Const |
|  | 9.Classes |
|  | 10.Modules. |
|  | 11.for of loop |
|  | 12.Spread attributes |
|  | 13.Maps |
|  | 14.Sets,Static methods,Getters and Setters... |
|  |  |
|  | ` |
|  | **1.const and let:** |
|  | //let is the same as const in that both are blocked-scope. It means that the variable is only available within its scope. |
|  | //exmpLet: |
|  | let b = 100; |
|  | var d=50; |
|  | if (true) { |
|  | let a = 60; //block scope not available outside this block |
|  | var c = 10; //function Scope available outside this block |
|  | console.log(a); // 60 |
|  | console.log(b); // 100 |
|  | console.log(d); // 50 |
|  | } |
|  | console.log(c); // 10 |
|  | //console.log(b); // 100 |
|  | //console.log(a); // ReferenceError: a is not defined |
|  | //Constexmp: |
|  |  |
|  | //Const is used to assign a constant value to the variable. And the value cannot be changed. Its fixed. |
|  |  |
|  | const x = 50; |
|  | //x = 60; // shows error. You cannot change the value of const. |
|  | const y = "Constant variable"; |
|  | //y = "Assigning new value"; // shows error. |
|  |  |
|  | **2.Arrow Functions:** |
|  |  |
|  | //Functions in ES6 have changed a bit. I mean the syntax. |
|  |  |
|  | // Old Syntax |
|  | function oldOne() { |
|  | console.log("Hello World..!"); |
|  | } |
|  | // New Syntax |
|  | var newOne = () => { |
|  | console.log("Hello World..!"); |
|  | } |
|  | //The new syntax may be confusing a little bit. But I will try to explain the syntax. |
|  |  |
|  | //There are two parts of the syntax. |
|  |  |
|  | //1.var newOne = () 2.=> {} |
|  |  |
|  | `The first part is just declaring a variable and assigning the function (i.e) () to it. It just says the variable is actually a function. |
|  |  |
|  | Then the second part is declaring the body part of the function. The arrow part with the curly braces defines the body part.` |
|  |  |
|  | //Another example with parameters. |
|  |  |
|  | let NewOneWithParameters = (p, q) => { |
|  | console.log(p+q); // 30 |
|  | } |
|  | NewOneWithParameters(10, 20); |
|  |  |
|  | **3.Default Parameters**: |
|  |  |
|  | `Default parameters are parameters which are given by default while declaring a function. |
|  | But it’s value can be changed when calling the function.` |
|  |  |
|  | Example: |
|  |  |
|  | let Func = (a, b = 10) => { |
|  | return a + b; |
|  | } |
|  | Func(20); // 20 + 10 = 30 |
|  | //In the above example, we are passing only one parameter. The function makes use of the default parameter and executes the function. |
|  |  |
|  | example2: |
|  |  |
|  | Func(20, 50); // 20 + 50 = 70 |
|  | //In the above example, the function takes two parameters and the second parameter replaces the default parameter. |
|  |  |
|  | example3: |
|  |  |
|  | let NotWorkingFunction = (a = 10, b) => { |
|  | return a + b; |
|  | } |
|  | NotWorkingFunction(20); // NAN. Not gonna work. |
|  | `When you are calling the function with parameters they get assigned in the order. (i.e) the first value gets assigned to the first parameter and the second value gets assign to the second parameter and so on.. |
|  |  |
|  | In the above example, the value 20 gets assigned to parameter ‘a’ and ‘b’ is not having any value. So we are not getting any output. |
|  |  |
|  | But,` |
|  |  |
|  | NotWorkingFunction(20, 30); // 50; |
|  |  |
|  | **4.For of loop** |
|  |  |
|  | `for..of is very similar to for..in with slight modification. |
|  |  |
|  | for..of iterates through list of elements (i.e) like Array and returns the elements (not their index) one by one.` |
|  |  |
|  | let arr = [2,3,4,1]; |
|  | for (let value of arr) { |
|  | console.log(value); |
|  | } |
|  | Output: |
|  | 2 |
|  | 3 |
|  | 4 |
|  | 1 |
|  |  |
|  | **5.Spread attributes** |
|  |  |
|  | //Spread attributes help to spread the expression as the name suggests. In simple words, it converts a list of elements to an array and vice versa. |
|  |  |
|  | //Example without spread attributes: |
|  |  |
|  | let SumElements = (arr) => { |
|  | console.log(arr); // [10, 20, 40, 60, 90] |
|  | let sum = 0; |
|  | for (let element of arr) { |
|  | sum += element; |
|  | } |
|  | console.log(sum); // 220. |
|  | } |
|  | SumElements([10, 20, 40, 60, 90]); |
|  | //Above example is straightforward. We are declaring a function to accept array as parameter and returning its sum. Its simple. |
|  |  |
|  | //Now consider the same example with spread attributes |
|  |  |
|  | let SumElements = (...arr) => { // ...arr is a spread operator |
|  | console.log(arr); // [10, 20, 40, 60, 90] |
|  | let sum = 0; |
|  | for (let element of arr) { |
|  | sum += element; |
|  | } |
|  | console.log(sum); // 220. |
|  | } |
|  | SumElements(10, 20, 40, 60, 90); // Note we are not passing array here. Instead we are passing the elements as arguments. |
|  | //In the above example, the spread attribute converts the list of elements (i.e) the parameters to an array. |
|  |  |
|  | Example2: |
|  |  |
|  | Math.max(10, 20, 60, 100, 50, 200); // returns 200. |
|  | //Math.max is a simple method that returns the maximum element from given list. It doesn’t accept an array. |
|  |  |
|  | let arr = [10, 20, 60]; |
|  | Math.max(arr); // Shows error. Doesn't accept an array. |
|  | //So lets use our savior. |
|  |  |
|  | let arr = [10, 20, 60]; |
|  | Math.max(...arr); // 60 |
|  | //In the above example, the spread attribute converts the array to list of elements. |
|  |  |
|  | **6.Maps** |
|  |  |
|  | `Map holds key-value pairs. It’s similar to an array but we can define our own index. And indexes are unique in maps.` |
|  | Example: |
|  |  |
|  | var NewMap = new Map(); |
|  | NewMap.set('name', 'John'); |
|  | NewMap.set('id', 2345796); |
|  | NewMap.set('interest', ['js', 'ruby', 'python']); |
|  | NewMap.get('name'); // John |
|  | NewMap.get('id'); // 2345796 |
|  | NewMap.get('interest'); // ['js', 'ruby', 'python'] |
|  |  |
|  | //I think the above example is self explanatory. |
|  |  |
|  | //Other interesting features of Maps are all indexes are unique. And we can use any value as key or value. |
|  |  |
|  | Example: |
|  |  |
|  | var map = new Map(); |
|  | map.set('name', 'John'); |
|  | map.set('name', 'Andy'); |
|  | map.set(1, 'number one'); |
|  | map.set(NaN, 'No value'); |
|  | map.get('name'); // Andy. Note John is replaced by Andy. |
|  | map.get(1); // number one |
|  | map.get(NaN); // No value |
|  | //Other useful methods used in Map: |
|  |  |
|  | var map = new Map(); |
|  | map.set('name', 'John'); |
|  | map.set('id', 10); |
|  | map.size; // 2. Returns the size of the map. |
|  | map.keys(); // outputs only the keys. |
|  | map.values(); // outputs only the values. |
|  | for (let key of map.keys()) { |
|  | console.log(key); |
|  | } |
|  | Output: |
|  | name |
|  | id |
|  | `In the above example, map.keys() returns the keys of the map but it returns it in Iterator object. It means that it can’t be displayed as it is. It should be displayed only by iterating.` |
|  |  |
|  | example2: |
|  |  |
|  | var map = new Map(); |
|  | for (let element of map) { |
|  | console.log(element); |
|  | } |
|  | Output: |
|  | ['name', 'John'] |
|  | ['id', 10] |
|  | //The above example is self explanatory. The for..of loop outputs the key-value pair in array. |
|  |  |
|  | //We can optimise it a little bit. |
|  |  |
|  | var map = new Map(); |
|  | for (let [key, value] of map) { |
|  | console.log(key+" - "+value); |
|  | } |
|  | Output: |
|  | name - John |
|  | id - 10 |
|  |  |
|  | //7.Sets |
|  |  |
|  | //Sets are used to store the unique values of any type. Simple..! |
|  |  |
|  | Example: |
|  |  |
|  | var sets = new Set(); |
|  | sets.add('a'); |
|  | sets.add('b'); |
|  | sets.add('a'); // We are adding duplicate value. |
|  | for (let element of sets) { |
|  | console.log(element); |
|  | } |
|  | Output: |
|  | a |
|  | b |
|  | `Note that no duplicate values are displayed. Unique values are displayed. |
|  |  |
|  | And also note that sets are iterable objects. We have to iterate through the elements to display it.` |
|  |  |
|  | //Other useful methods: |
|  |  |
|  | var sets = new Set([1,5,6,8,9]); |
|  | sets.size; // returns 5. Size of the size. |
|  | sets.has(1); // returns true. |
|  | sets.has(10); // returns false. |
|  | 'In the above example, size is self-explanatory. There is another method ‘has’ which return a boolean value based on whether the given element is present in the set or not.' |
|  |  |
|  | **8.Static methods** |
|  |  |
|  | `Most of you have already heard about static methods. Static methods are introduced in ES6. And it is pretty much easy to define it and use it.` |
|  |  |
|  | Example: |
|  |  |
|  | class Example { |
|  | static Callme() { |
|  | console.log("StaticMethod"); |
|  | } |
|  | } |
|  | Example.Callme(); |
|  | Output: |
|  | StaticMethod |
|  | //Note that I didn’t use the keyword ‘function’ inside Class. |
|  |  |
|  | //And I can call the function without creating any instance for the class. |
|  |  |
|  | //10.Getters and Setters |
|  |  |
|  | `Getters and setters and one of the useful feature introduced in ES6. It will come in handy if you are using classes in JS.` |
|  |  |
|  | //Example without getters and setters: |
|  |  |
|  | class People { |
|  | constructor(name) { |
|  | this.name = name; |
|  | } |
|  | getName() { |
|  | return this.name; |
|  | } |
|  | setName(name) { |
|  | this.name = name; |
|  | } |
|  | } |
|  | let person = new People("JonSnow"); |
|  | console.log(person.getName()); |
|  | person.setName("Dany"); |
|  | console.log(person.getName()); |
|  | Output: |
|  | JonSnow |
|  | Dany |
|  | //I think the above example is self-explanatory. We have two functions in class People that helps to set and get the name of the person. |
|  |  |
|  | //Example with getters and setters |
|  |  |
|  | class People { |
|  | constructor(name) { |
|  | this.name = name; |
|  | } |
|  | get Name() { |
|  | return this.name; |
|  | } |
|  | set Name(name) { |
|  | this.name = name; |
|  | } |
|  | } |
|  | let person = new People("Jon Snow"); |
|  | console.log(person.Name); |
|  | person.Name = "Dany"; |
|  | console.log(person.Name); |
|  | `In the above example, you can see there are two functions inside class People with ‘get’ and ‘set’ properties. The ‘get’ property is used to get the value of the variable and ‘set’ property is used to set the value to the variable. |
|  |  |
|  | And you can see that getName function is called without parenthesis. And setName function is called without parenthesis and it’s just like assigning a value to the variable.` |
|  |  |
|  | //12.Template Literals |
|  | `Template literals or template strings are pretty cool. We don’t have to use the plus (+) operator to concatenate strings, or when we want to use a variable inside a string.` |
|  |  |
|  | var p = { |
|  | name: 'kumar', age: '27', |
|  | }; |
|  | oldsyntax: |
|  | // STRING CONCATENATION |
|  | console.log('Hi, I\'m ' + p.name + 'And my age "' + p.age + '".'); |
|  |  |
|  | newsyntax: |
|  | // TEMPLATE LITERALS |
|  | console.log(`Hi, I'm ${p.name} And my age "${p.age}".`); |
|  |  |
|  | **9.promises:** |
|  |  |
|  | `A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred). |
|  | A promise may be in one of 3 possible states: fulfilled, rejected, or pending. |
|  | Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.` |
|  |  |
|  | //How Promises Work |
|  | `A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states: |
|  |  |
|  | Fulfilled: onFulfilled() will be called (e.g., resolve() was called) |
|  | Rejected: onRejected() will be called (e.g., reject() was called) |
|  | Pending: not yet fulfilled or rejected` |
|  |  |
|  | `Promises following the spec must follow a specific set of rules: |
|  |  |
|  | A promise or “thenable” is an object that supplies a standard-compliant .then() method. |
|  | A pending promise may transition into a fulfilled or rejected state. |
|  | A fulfilled or rejected promise is settled, and must not transition into any other state. |
|  | Once a promise is settled, it must have a value (which may be undefined). That value must not change.` |
|  |  |
|  | const promise = new Promise((resolve, reject) => { |
|  | //asynchronous code goes here |
|  | }); |
|  |  |
|  | `promise.then( |
|  | onFulfilled?: Function, |
|  | onRejected?: Function |
|  | ) => Promise` |
|  |  |
|  | ex: |
|  |  |
|  | const promise = new Promise((resolve, reject) => { |
|  | const request = new XMLHttpRequest(); |
|  |  |
|  | request.open('GET', 'https://api.icndb.com/jokes/random'); |
|  | request.onload = () => { |
|  | if (request.status === 200) { |
|  | resolve(request.response); // we got data here, so resolve the Promise |
|  | } else { |
|  | reject(Error(request.statusText)); // status is not 200 OK, so reject |
|  | } |
|  | }; |
|  | //enough for this |
|  |  |
|  | request.onerror = () => { |
|  | reject(Error('Error fetching data.')); // error occurred, reject the Promise |
|  | }; |
|  |  |
|  | request.send(); // send the request |
|  | }); |
|  |  |
|  | console.log('Asynchronous request made.'); |
|  |  |
|  | promise.then((data) => { |
|  | console.log('Got data! Promise fulfilled.'); |
|  | document.body.textContent = JSON.parse(data).value.joke; |
|  | }, (error) => { |
|  | console.log('Promise rejected.'); |
|  | console.log(error.message); |
|  | }); |
|  |  |
|  | **15. Multi-line Strings in ES6** |
|  | `Another yummy syntactic sugar is multi-line string. In ES5, we had to use one of these approaches:` |
|  |  |
|  | var roadPoem = 'Then took the other, as just as fair,\n\t' |
|  | + 'And having perhaps the better claim\n\t' |
|  | + 'Because it was grassy and wanted wear,\n\t' |
|  | + 'Though as for that the passing there\n\t' |
|  | + 'Had worn them really about the same,\n\t' |
|  |  |
|  | var fourAgreements = 'You have the right to be you.\n\ |
|  | You can only be you when you do your best.' |
|  | //While in ES6, simply utilize the backticks: |
|  |  |
|  | var roadPoem = `Then took the other, as just as fair, |
|  | And having perhaps the better claim |
|  | Because it was grassy and wanted wear, |
|  | Though as for that the passing there |
|  | Had worn them really about the same,` |
|  |  |
|  | var fourAgreements = `You have the right to be you. |
|  | You can only be you when you do your best.` |