ECMAScript6(ES6)

ITI - Assiut Branch

Eng. Sara Ali

Introduction

What Is ES6?

- JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.
- ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
- ES6 refers to version 6 of the ECMA Script programming language. ECMA Script is the standardized name for JavaScript. It is a major enhancement to the JavaScript language, and adds many more features intended to make large-scale software development easier.
- ECMAScript, or ES6, was published in June 2015. It was subsequently renamed to ECMAScript 2015.

New Features in ES6

- The let keyword
- The const keyword
- Default Parameters
- Arrow Functions
- Template Literals
- Spread Operator
- Rest Parameter
- String New Methods
- Array New Methods
- forEach Method
- Object Destructuring
- Array Destructuring
- Classes
- Set Objects
- Map Objects
- Generators
- Modules
- Proxy
- Promises
- JavaScript Fetch API
- JavaScript Async/Await

The let Keyword

The let keyword was introduced in ES6 (2015).

- Variables defined with let cannot be Redeclared.

```
var x = "Hello";
var x = 0;
console.log(x); // 0
```

```
let x = "Hello";
let x = 0;
console.log(x);
// SyntaxError: 'x' has
already been declared
```

- Variables defined with let must be Declared before use.

```
console.log(x);
var x = 1;
// undefined
```

```
console.log(x);
let x = 1;
/*SyntaxError: cannot access
'x' before initialization*/
```

The let Keyword(Cont.)

- Variables defined with let have Block Scope.

```
function test(){
    var x = 1;
    if(true){
        var x = 2;
        console.log(x); //2
    }
console.log(x); //2
}
test();
```

```
function test(){
    let x = 1;
    if(true){
        let x = 2;
        console.log(x); //2
    }
console.log(x); //1
}
test();
```

- Variables defined with let don't create properties in window object

```
var x = "Hello";
let y = "ES6";
console.log(window.x); //Hello
console.log(window.y); //undefined
```

The Const Keyword

The const keyword was introduced in ES6 (2015).

- Variables defined with const cannot be Redeclared.
- Variables defined with const have Block Scope.
- Variables defined with const don't create properties in window object
- Variables defined with const cannot be Reassigned.

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
```

The Const Keyword(Cont.)

- JavaScript const variables must be assigned a value when they are declared.

```
const PI; //SyntaxError: Missing initializer in const
declaration
PI = 3.14159265359;
```

When to use JavaScript const?

- Use const when you declare:
  - A new Array
  - A new Object
  - A new Function
  - A new RegExp

The Const Keyword(Cont.)

Constant Objects and Arrays

- The keyword const does not define a constant value. It defines a constant reference to a value.
- Because of this you can not reassign a constant array or reassign a constant object. But you can:
  - Change the elements of constant array
  - Change the properties of constant object

The Const Keyword(Cont.)

Constant Arrays

```
// You can create a constant array:
const Arr = ["a", "b", "c"];

// You can change an element:
Arr[0] = "d";

// You can add an element:
Arr.push("e");

// You can not reassign the array:
Arr = ["Toyota", "Volvo", "Audi"]; // ERROR
```

The Const Keyword(Cont.)

Constant Objects

```
// You can create a const object:
const CAR = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
CAR.color = "red";

// You can add a property:
CAR.owner = "Johnson";

// You can NOT reassign the object:
CAR = {type:"Volvo", model:"EX60", color:"red"}; // ERROR
```

Default Parameter Values

- ES6 allows function parameters to have default values.

```
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
myFunction(5); // will return 15
```

Arrow Functions

- Arrow functions allow us to write shorter function syntax:

```
let x = function() {
  return "Hello";
}
```

```
let x =  () => {
  return "Hello";
}
```

- It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword:

```
let x = () => "Hello World!";
```

Arrow Functions(Cont.)

- If you have parameters, you pass them inside the parentheses:

```
let x = (val) => "Hello " + val;
```

- In fact, if you have only one parameter, you can skip the parentheses as well:

```
let x = val => "Hello " + val;
```

What About this?

- In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.
- With arrow functions the this keyword always represents the object that defined the arrow function.

Template Literals

- **Template Literals** use back-ticks (``) rather than the quotes ("") to define a string:

```
let text = `Hello World!`;
```

- With **template literals**, you can use both single and double quotes inside a string:

```
let text = `He's often called "Johnny"`;
```

- **Template literals** allows multiline strings:

```
let text =
`The quick
brown fox
jumps over
the lazy dog`;
```

Template Literals(Cont.)

- **Template literals** provide an easy way to interpolate variables and expressions into strings. The method is called string interpolation.

```
let firstName = "John";
let lastName = "Doe";

let text = `Welcome ${firstName}, ${lastName}!`;
```

- **Template literals** allow expressions in strings:

```
let  gender = "Female";

let  x = `Hello It's Me, ${gender==="Female"? "Mrs" :
"Mr"}  Esraa.`;
```

Spread Operator

- allows iterables (arrays / objects / strings) to be **expanded** into single arguments/elements.

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];
document.write(numbersCombined);
```

```
function sum(x,y,z){
      return x+y+z;
}

const myNumbers = [1,2,3]

console.log(sum(...myNumbers)); //6
```

Rest parameter

- collects all **remaining** elements into an **array**.

```
function add(x, y) {
        return x + y;
}

console.log(add(1, 2, 3, 4, 5)); // returns 3
only the fist two arguments will be counted.
```

- With rest parameters we can gather any number of arguments into an array and do what we want with them. So we can re-write the add function like this:

Rest parameter(Cont.)

```
function add (...args) {
    let result = 0;

    for (let arg of args)
    result += arg;

    return result
}
add (1); // returns 1
add (1,2); // returns 3
add (1, 2, 3, 4, 5); // returns 15
```

- We can separately define the first arguments, and the rest of the arguments in the function call (no matter how many they are) will be collected into an array by the rest parameter.

Rest parameter(Cont.)

```
function xyz(x, y, ...z) {
   console.log(x, ' ', y); // hey hello

   console.log(z); // ["wassup", "goodmorning", "hi", "howdy"]
   console.log(z[0]); // wassup
   console.log(z.length); // 4
}

xyz("hey", "hello", "wassup", "goodmorning", "hi", "howdy")
```

- **Note:** Rest parameters have to be at the **last argument**. This is because it collects all remaining/excess arguments into an array.
- Since the rest parameter gives us an array, we can use array methods like Array.find e.t.c.

String New Methods

```
var text = " Hello world, welcome to ITI";
```

Name	Description	Example
startsWith()	The startsWith() method returns true if a string begins with a specified value, otherwise false.	text.startsWith("Hello"); // Returns true text.startsWith("Hello", 5); // Returns false
endsWith()	The endsWith() method returns true if a string ends with a specified value, otherwise false.	text.endsWith("ITI") // Returns true text.endsWith("ITI", 23) // Returns false
Includes()	The includes() method returns true if a string contains a specified value, otherwise false.	text.includes("world") // Returns true text.includes("world", 11) // Returns false

Array New Methods

Name	Description	Example
Array.from()	The Array.from() method returns an Array object from any object with a length property or any iterable object. Array.from(Iterable, MapFn)	let text = "ABCDEFG" Array.from(text); // Returns [A,B,C,D,E,F,G]

Array keys()	The keys() method returns an Array Iterator object with the keys of an array.	const fruits = ["Banana", "Orange", "Apple", "Mango"];
		const keys = fruits.keys();
		
		let text = "";
		for (let x of keys) {
		  text += x ;
		}
		// 0123

Array New Methods(Cont.)

```
const numbers= [4, 9, 16, 25, 29];
```

Name	Description	Example
find()	The find() method returns the value of the first array element that passes a test function.	let first = numbers.find(num => num > 18); // 25
findIndex()	The findIndex() method returns the index of the first array element that passes a test function.	let first = numbers.findIndex(num=>num>18); // 3

fill()	The fill() method fills specified elements in an array with a value.
	
	array.fill(value, start, end)
		numbers.fill(100);
		// [100,100,100,100,100]
		
		numbers.fill(100, 2, 4);
		// [4,9,100,100,29]

Array New Methods(Cont.)

Name	Description	Example

map()	- map() creates a new array from calling a function for every array element.
	- map() is an ECMAScript5 (ES5) feature.
		const numbers = [65, 44, 12, 4];
		const newArr = numbers.map(num => {return num * 10});
		// [650,440,120,40]

filter()	- The filter() method creates a new array filled with elements that pass a test provided by a function.
	- filter() is an ECMAScript5 (ES5) feature.
		const ages = [32,33,12,40,20];
		ages.filter(age=>{return age >30});
		// [32, 33, 40]

copyWithin()	The copyWithin() method copies array elements to another position in the array. array.copyWithin(target, start default 0, end default array.length)	let letters = ["A", "B", "C", "D", "E"] ; letters.copyWithin(1, 3, 5) // ["A", "D", "E", "D", "E"]

forEach Method

- forEach() method calls a function for each element

```
const numbers = [65, 44, 12, 4];

numbers.forEach(myFunction)

function myFunction(item, index, arr)
{
  arr[index] = item * 10;
}
```

```
numbers.forEach((item, index, arr))=>{
        arr[index] = item * 10;
})
```

Object Destructuring

- object destructuring that assigns properties of an object to individual variables.
- Here is the old way of assigning object properties to a variable:

```
let person = {
      firstName: 'John',
      lastName: 'Doe'
};

let firstName = person.firstName;
let lastName = person.lastName;
```

Object Destructuring(Cont.)

- ES6 introduces the object destructuring syntax that provides an alternative way to assign properties of an object to variables.

```
let person = {
   firstName: 'John',
   lastName: 'Doe'
};

let { firstName, lastName} = person;
or
let { firstName: fname, lastName: lname } = person;
```

Object Destructuring(Cont.)

- When you assign a property that does not exist to a variable using the object destructuring, the variable is set to undefined.

```
let { firstName, lastName, middleName } = person;

console.log(middleName); // undefined

let { firstName, lastName, middleName = 'Default',
currentAge: age = 18 } = person;

console.log(middleName); // Default
console.log(age); // 18
```

Object Destructuring(Cont.)

- A function may return an object or null in some situations. For example:

```
function getPerson() {
        return null;
}

let { firstName, lastName } = getPerson();
// TypeError: Cannot destructure property 'firstName'
of 'getPerson(...)' as it is null.

let { firstName, lastName } = getPerson() || {};
Console.log(firstName); // undefined
```

Object Destructuring(Cont.)

- Nested object destructuring
- Assuming that you have an employee object which has a name object as the property:

```
let employee = {
    id: 1001,
    name: {
        firstName: 'John',
        lastName: 'Doe'
    }
};
let { id, name} = employee;
document.write(name); // [object Object]
document.write(name.firstName); // John

let { id, name:{firstName}} = employee;
document.write(firstName); // John
document.write(lastName); // ReferenceError: lastName is not defined
```

Array Destructuring

- Assuming that you have a function that returns an array of numbers as follows:

```
function getScores() {
        return [70, 80, 90];
}

let [x, y, z] = getScores();
console.log(x); // 70
console.log(y); // 80
console.log(z); // 90
```

- The variables x, y and z will take the values of the first, second, and third elements of the returned array.
- Note that the square brackets [] look like the array syntax but they are not.

Array Destructuring(Cont.)

- If the getScores() function returns an array of two elements, the third variable will be undefined, like this:

```
function getScores() {
        return [70, 80];
}

let [x, y, z] = getScores();
console.log(x); // 70
console.log(y); // 80
console.log(z); // undefined
```

- In case the getScores() function returns an array that has more than three elements, the remaining elements are discarded

Array Destructuring(Cont.)

- It’s possible to take all remaining elements of an array and put them in a new array by using the rest syntax (...):

```
function getScores() {
        return [70, 80, 90, 100];
}
let [x, y ,...args] = getScores();
console.log(x); // 70
console.log(y); // 80
console.log(args); // [90, 100]
```

- If we only want x and args.

```
let [x, ,...args] = getScores();
```

Array Destructuring(Cont.)

- If you want to set a default value

```
function getScores() {
        return [70, 80];
}
let [x, y, z = 'Default'] = getScores();
console.log(x); // 70
console.log(y); // 80
console.log(z); // Default
```

- Swap two variables

```
var book = "video";
var video = "book";

[book, video] = [video, book];
console.log(book,video); // book video
```

JavaScript

THANK YOU