ECMAScript 2015 (ES6)

softserve

Agenda

- babel.js
- let and const
- template Literals
- arrow functions
- spread operator
- rest parameters
- destructuring
- symbol
- iterator
- Collections. Set & Map

What is ECMAScript 6?

JavaScript language standard: **EcmaScript** (ECMA-262)

1995 to 2014: ES1 (1997), ES2 (1998), ES3 (1999) and ES5 (2009).

From 2015 - a new significantly updated language standard (**ES2015**) and new versions should be released annually.

Web developers now do not have to rely on the support of all the features of the modern language standard in browsers, but use special tools that can convert the code written in the modern language (say ES2016) to some previous one (eg ES5).

What is new in ECMASCRIPT 2015?

- Scoping (Block-Scoped Variables, Block-Scoped Variables)
- Arrow Functions
- Extended Parameter Handling (Default Parameter Value, Rest Parameter, Spread Operator)
- Template Literals
- Enhanced Regular Expression
- Destructuring Assignment
- Modules
- Classes
- Symbol Type
- Iterators
- Generators
- Promises
- Map/Set & WeakMap/WeakSet

babel.js

Babel.JS is a transpiler that rewrites the code on ES-2015 to code based on the previous ES5 standard.

Installation:

```
npm install --save-dev babel-cli babel-preset-env
```

Launching:

```
npm run babel
```

src/test.js:

```
const message = `Hello World`;
console.log(message);
```

npm run babel

dist/test.js:

```
"use strict";

var message = "Hello World";
console.log(message);
```

softserve

let and const

In ES6, the keywords **let** and **const** are used instead of **var** to **declare variables.**

Let opportunities:

1) new block scope - block {...}
2) the variable is visible ONLY after the declaration
3) a new variable is created in the loop after each iteration

Const opportunities:

- it's just a constant (const PI = 3.14159265359;)

1) new block scope - block {...}

A variable declared through ***var*** is visible **everywhere in a function**

The variable declared through ***let*** is visible only **within the framework of the block** {...} in which it is declared

```
if (true) {
    var name1 = "Ben";
    let name2 = "Nick";
}
console.log(name1); // Ben
console.log(name2); // ReferenceError: name2 is not defined
```

softserve

let and const

2) the variable is visible ONLY after the declaration

Variables declared through **var exist** even before the declaration (hoisting).
Variables declared via **let do not exist** before declaration:

```
alert(name1); // undefined
var name1 = "Ben";
```

```
alert(name1); // ReferenceError: name1 is not defined
let name1 = "Ben";
```

3) a new variable is created in the loop after each iteration

The *var* i declaration creates one variable for all iterations of the loop, so all functions close the same variable.
The declaration *let* i creates for each repetition of a block in a loop its variable, which the function receives from the closure.

```
var array = [];
for (var i = 0; i < 10; i++) {
 array[i] = function () {
      alert(i);
   };
}
array[0](); // 10
array[4](); // 10
```

```
let array = [];
for (let i = 0; i < 10; i++) {
 array[i] = function () {
      alert(i);
   };
}
array[0](); // 0
array[4](); // 4
```

softserve

Template Literals

ES6 Added New Type of Quotation Marks for Strings:

```
const newString = `text in backtick`;
```

The **main differences** from double "..." and single '...' quotes:

- You can insert expressions using **${...}**:

```
const name = `John`;
const job = `engineer`;
const str = `${name} works at factory as an ${job}`;
console.log(str); // "John works at factory as an engineer"

const salary = 900;
const bonus = 400;
const income = `Your monthly income will be ${salary + bonus}`;
console.log(income);   // Your monthly income will be 1300
```

- Now you can do a **line feed**:

```
console.log(`Example
   multiline
      lines`);
```

```
Example
    multiline
       lines
```

softserve

Arrow functions

Arrow functions ("=>") are a new short syntax for creating functions in ES2015.

You **don't need** the **function keyword**, the **return keyword**, and the **curly brackets**.

ES6 arrow function:

```
let reflect = value => value;
```

ES5 function:

```
let reflect = function(value) {
    return value;
};
```

- Arrow functions are **not hoisted**. They must be defined **before** they are used.
- If there are **several parameters**, they are **wrapped in parentheses**

Arrow-function with several arguments

```
const sum = (a,b) => a + b;
// ES5 analog
// let sum = function(a, b) { return a + b; };
alert( sum(1, 2) ); // 3
```

Arrow-function as a callback

```
const arr = [5, 8, 3];
const sorted = arr.sort( (a,b) => a - b );
alert(sorted); // 3, 5, 8
```

softserve

Arrow functions do not have their own this

- An important difference in the behavior of the arrow functions with respect to the functional expressions is that they do not attach their meaning to this and refer to this declared in the parent function.
- This makes the code much more compact in many cases when we need to access the object instance.

Function ES5

```
function Person() {
    let self = this; // here we catch this
    this.age = 0;

    setInterval(function growUp() {
        self.age++;
    }, 1000);
}
let p = new Person();
```

Arrow function ES6

```
function Person() {
    this.age = 0;

    setInterval(() => {
// |this| properly refers to the person object
        this.age++;
    }, 1000);
}
let p = new Person();
```

softserve

Spread operator

The **spread operator "…"** allows you to extend expressions in places where multiple arguments are provided

```
const numbers = [11, 8, 44, 87];
const minimum = Math.min(...numbers);
console.log(minimum); // 8
```

Spread will expand the array in place and pass the elements in as if it were a comma separated list.

Using the spread operator to concat

You can also use the spread operator to concatenate arrays together! Since spread expands arrays, we can expand arrays in arrays!

```
const arr1 = [7, 3, 9, 12, 21];
const arr2 = [22, 7, 15, 18, 33];
const concatArray = [...arr1, ...arr2];
console.log(concatArray); // [7, 3, 9, 12, 21, 22, 7, 15, 18, 33]
```

softserve

rest parameters

If more parameters than declared are passed to the function, all remaining parameters can be obtained using the operator "...":

```
const func = (arg1, ...rest) => {
   console.log(arg1);
   console.log(rest);
}
func("First", "Second", "Third", "Fourth"); // "First"
                                            // ["Second", "Third", "Fourth"]
```

Rest will get all the parameters after the first.

The operator **...** can also be used when calling a function to pass an **array of parameters as a list**:

```
function func(arg1, arg2, arg3) {
   console.log(arg1 + " " + arg2 + " " + arg3);
}
const data = ["First", "Second", "Third", "Fourth"];
func(data);    // "First, Second, Third, Fourth undefined undefined"
func(...data); // "First Second Third"
```

softserve

Destructuring

Another important innovation is **destructuring**, which is a syntactic construction of language that allows us to derive values from arrays or objects into separate variables.
The syntax for destructing is similar to literal expressions, however, on the left side of the assignment statement, the elements to be obtained from the array or object indicated on the right side are specified.

```
const colors = ["red", "green", "blue"];
const [firstColor, secondColor] = colors;
console.log(firstColor);       // "red"
console.log(secondColor);  // "green"
```

You can also **omit items** in the destructuring template and give the name of the variable you need. For example, if you need a third element of an array, there is no need to substitute names for the first and second elements.

```
const colors = ["red", "green", "blue"];
const [ ,  , thirdColor] = colors;
console.log(thirdColor); // " blue "
```

Destructuring with spread operator

If you need to get the subsequent values of the array, but you don't know how many there will be, you can add another parameter that will get everything else using the operator **...** (**ellipsis**):

```
const [firstColor, ...rest] = ["red", "green", "blue", "brown"];
console.log(firstColor); // "red"
console.log(rest);       // ["green", "blue", "brown"]
```

The **rest** value will be an array of the remaining elements of the array. Instead of rest, you can use another variable name

Please note that the ellipsis must be the last element in the list.

Destructuring. Objects

Object Destructuring Syntax uses the object literal on the left side of the assignment operation:

```
const direction = {
        type: "web",
        name: "JavaScript"
};
const { type, name } = direction;   // const type = direction.type;
                                    // const name = direction.name;
console.log(type);   // "web"
console.log(name);   // "JavaScript"
```

Destructuring can be **combined and put into each other** as you please:

```
const cities = {
   first: "Lviv",
   second: "Kyiv",
   third: {p1:"Berlin", p2: "Hamburg"}
};
const {first: f, second: s, third: {p1, p2}} = cities;
console.log(f); // "Lviv"
console.log(s); // "Kyiv"
console.log(p1); // "Berlin"
console.log(p2); // "Hamburg"
```

softserve

Destructuring in functions parameters

If a function receives an object, it can immediately break it into variables:

```
function func( {first, second, third} ) {
   console.log(first + " " + second + " " + third);
}
const cities = {
   first: "Lviv",
   second: "Kyiv",
   third: "Dnipro"
};
func(cities); // "Lviv Kyiv Dnipro"
```

Symbol

- Symbols - a new primitive type, designations for unique identifiers.
- All symbols are unique. Symbols with the same name - different Symbols.

Syntax

```
let sym = Symbol();
alert( Symbol("name") === Symbol("name") );  // false
```

"name“ – character description (name)

- Sometimes, on the contrary, we want characters with the same name to be one entity. To do this, there is a global register of characters available through the **Symbol.for("name")** method. For a global character, you can get a name (key) by calling **Symbol.keyFor (sym)**.

```
let data1 = Symbol.for("data");
let data2 = Symbol.for("data");
console.log(data1 === data2); // true
```

Symbol. "Hidden" properties

Symbols allow you to create "hidden" properties of objects that cannot be accidentally accessed and overwrite them from other parts of the program.

```
let key = Symbol("key");
let person = {
   position: "webDev",
   experience: 3,
   [key]: 100
};
for (let props in person) {
   console.log(props);          // position, experience
}
console.log(person[key]); // 100
```

The symbols are ignored by the **for ... in** loop

Symbol usage

Symbols have two main uses:

- **Hidden properties of objects**. If we want to add a property to an object that "belongs" to another script or library, we can create a symbol and use it as a key. The character property will not appear in for..in, so it will not be inadvertently processed along with others. Also, it will not be modified by direct access, since the other script does not know about our symbol. Thus, the property will be protected from accidental overwriting or use..

So, using symbolic properties, we can hide something we need, but that others should not see.

- There are many system symbols used inside JavaScript, available as Symbol. We can use them to **change the built-in behavior of a number of objects**. For example, in later chapters we will use Symbol.iterator for iterators.

Iterator

Iterated (or **iterable**) objects is a concept that allows you to use any object in a **for..of** loop.

Of course, **arrays** are iterable objects. But there are many other built-in iterable objects, like **strings**.

```
const str = "DOM";
for (let elem of str) {     // D
   console.log(elem);       // O
}                           // M
```

Unlike arrays, iterated objects **may not have length.**

The **for ... of** construct at the beginning of its execution **automatically calls Symbol.iterator.**

You can create your own iterator.

Explicit Iterator Call

You can iterate over a string in the same way as a for..of loop, but manually, by direct calls to **Symbol.iterator**.

```
let str = "DOM";
 let iterator = str[Symbol.iterator]();
 while (true) {
   let res = iterator.next();
   if (res.done) break;
   console.log(res.value);
 }
```

Output:

D

O

M

The **next()** method should return an object with properties with each call:

- **value** - the next value of the object being searched
- **done** - false if there are more values, and true - when the search is over

softserve

Collections

softserve

Collections. Set

Previously, we used complex data structures such as arrays (ordered collections) and objects (named collections). However, there are situations when their capabilities may not be sufficient to meet the challenges.

Set represent a data structure that can only **store unique values**. In JavaScript, set functionality defines a **Set object**. To create a set, use the constructor of this object:

```
const newSet = new Set( [args] );
```

Basic methods of the Set object:

- **set.add(value)** – adds a value (**if it already exists, then does nothing**), returns the same set object.
- **set.delete(value)** – removes the value, returns true if value was in the set at the time of the call, otherwise false.
- **set.has(value)** – returns true if the value is present in the set, otherwise false.
- **set.clear()** – deletes all available values.
- **set.size** – returns the number of elements in the set.

Collections. Set. Iterator

```
const cities = new Set();
const kyiv = "Kyiv";
const rome = "Rome";
const berlin = "Berlin";
const madrid = "Madrid";
cities.add(kyiv);
cities.add(rome);
cities.add(berlin);
cities.add(madrid);
cities.add(kyiv);
console.log(cities.size); // 4
console.log(cities.has(berlin)); // true
// Iterator
for (let city of cities) {
  console.log(city); // Kyiv Rome Berlin Madrid
}
```

softserve

Collections. Map

Map (dictionary) represents a data structure where each element has a **key and a value**. The **keys** within the card are **unique**, that is, only one element can be associated with one key. The key may be an **arbitrary value**. To **create a map**, the **Map object** constructor is used:

```
const newMap = new Map( [args] );
```

Basic **methods** of the Map object:

- **map.set(key, value)** – writes the value *value* by *key*.
- **map.get(key)** – returns value by *key* or undefined if *key* is absent.
- **map.has(key)** – returns true if *key* is present in the collection, otherwise false.
- **map.delete(key)** – deletes an element by *key*.
- **map.clear()** – clears a collection of all elements.
- **map.size** – returns the current number of elements.

Collections. Map

Map example:

```
const map= new Map();
map.set("name", "Nicholas");
map.set("age",  25);
console.log(map.size);                  //      2
console.log(map.has("name"));           //      true
console.log(map.get("name"));           //      "Nicholas"
console.log(map.has("age"));            //      true
console.log(map.get("age"));            //      25
map.delete("name");
console.log(map.has("name"));           //      false
console.log(map.get("name"));           //      undefined
console.log(map.size);                  //      1
map.clear();
console.log(map.has("age"));            //      false
console.log(map.get("age"));            //      undefined
console.log(map.size);                  //      0
```

softserve

Map can use objects as keys.

Collections. Iteration over Map

There are 3 methods for iterating over a Map collection:

- **map.keys()** - returns an iterable object by keys,
- **map.values()** - returns an iterable object by values,
- **map.entries()** - returns an iterable object by pairs of the form [key, value], this option is used by default in for..of.

```
const recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // cucumber, tomatoes, onion
}
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}
for (let entry of recipeMap) { // the same as of recipeMap.entries()
  alert(entry); // cucumber,500 (and so on)
}
```

softserve

Useful links

https://understandinges6.denysdovhan.com/
https://css-tricks.com/lets-learn-es2015/
https://ponyfoo.com/articles/es6

softserve

THANKS

softserve