ES Next

(birth name: ES6)

Frontend Junior Program - 2021

Introduction

Some say, things need to be *improved*. Well then, here you are: the new version of JavaScript

I'd rather think: things need to be *adapted*. As the environment around us continuously changes, we'd need to acquire new knowledge, new expertise – not to *improve* ourself, but to adapt to the new conditions, to be able to live with our potential in the new world.

The JavaScript ecosystem has changed dramatically in the recent years – the language had to be changed as well.

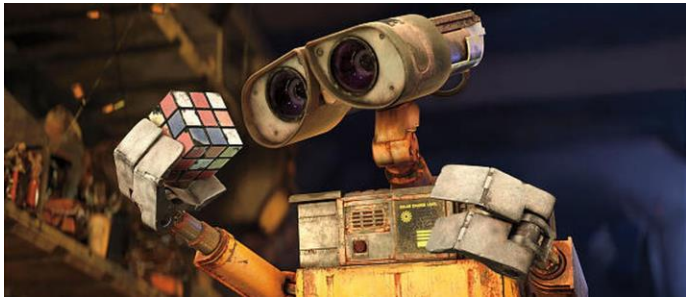
* * *

The largest batch of changes has been arrived with the ES6 (ES 2015), and after that the language receives new features gradually. While we have new releases year by year, we don't need to wait for the releases, nor for the browsers to adopt those, we can use immediately when transpilers (Babel and TypeScript) supports them.

Reference

Past lectures

ES6 basics	
let, const	*JS Intro*
default parameters	*JS Functions*
Set, Map	*JS Data Types*
ES6 methods	
find, repeat, includes, startsWith, endsWith	*JS Data Types*



our lectures already included a lot...

Upcoming lectures

Class, optional chaining	*Object Oriented Programming*
ES6 Modules	*JS Modules*
Promises	*Ajax*

still, new challenges are awaiting

- arrow functions
- spread syntax
- rest parameters
- array and object destructuring assignment
- shorthand properties and methods, computed property names

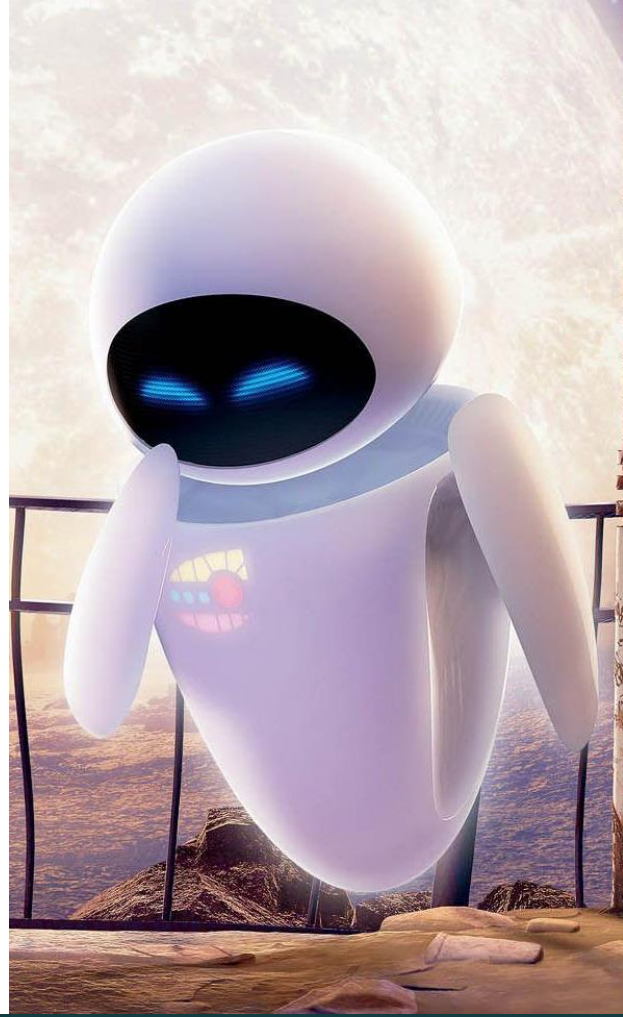
ES Next – for real?

There is one thing worth to mention:

The features we'll go through here are not some kind of exotic, never used peculiarities, interesting only for JavaScript aficionados, who are looking for special gems in the TC39 Proposals. It's is not that it would be wrong to do that - temporal will be especially important in future -, but these are actually used in projects on a daily basis.

We skipped these so far, because they will add a cognitive complexity to the code at first. A code fully packed with these are not just simply harder to understand, but it is very easy to tune up to an extremely cryptographic level.

These are like a very sharp chef knife: absolute necessities but should be used with care and expertise – you will see.



ARROW FUNCTIONS

Arrow functions

More precisely: arrow function expressions

It is important to realize that arrow functions are not the replacement of the function declarations, arrow functions are expressions. Basically, there are 2 differences between traditional and arrow function expressions: first, it is vastly more compact, and in some cases, it works differently (usually with OOP, we will deep dive into these only, when discussing the OOP).

As a consequence, arrow functions are targeted to use as a callback, mainly in functional programming.

traditional		*arrow function*
`function wallE() { }`	← a function declaration →	*there is no such a thing*
`const wallE = function() { };`	← function expression →	`const eve = () => { };`

Basic syntax

a parameter list, without a function name

an arrow, hence the name

```
> const eve = (status, message) => {
    return status + ": " + message;
  };

  eve("red", "Classified");

< "red: Classified"
```

otherwise, it is pretty much the same as a traditional function

Variations

full syntax →

```
let eve = (status, message) => {
  return status + ": " + message;
};
```

the *return* and the *brackets* can be skipped →

```
eve = (status, message) => status + ": " + message;
```

the return value will be the value of the expression

with *one parameter* only the parentheses can be skipped →

```
eve = status => status + "!";
```

returning an *object*, extra parenthesis are needed around the object →

```
eve = (status, message) => ({
  status: status,
  message: message
});
```

Why is it useful?

You will work with data all the time

Many times, the task is to transform data through multiple stages. These chains are usually long and utilizes complex functionalities.

Being able to focus on what really matters, is very important.

traditional callback functions →

```
> const heroes = [{
    kind: "human",
    name: "Captain B. McCrea"
  }, {
    kind: "robot",
    name: "Wall-E"
  }, {
    kind: "robot",
    name: "EVE"
  }];

  heroes.filter(function(hero) {
    return hero.kind === "robot";
  }).map(function(hero) {
    return hero.name;
  });
<· ▶ (2) ["Wall-E", "EVE"]
```

does the same, but with much less noise →

```
> heroes
    .filter(hero => hero.kind === "robot")
    .map(hero => hero.name);
<· ▶ (2) ["Wall-E", "EVE"]
```

Preventing bugs

Let me emphasize a subtle detail here

With having return statement in functions, it is possible that there will be other code lines which could have side effects. Without a *return*, you should be really wicked* to make side effects (for example, changing state variables).

When reviewing code parts *without returns* it is possible to focus on the return value only. When you need to review lots of code, the difference in effort could be night and day.

that's it, we have to focus on bugs

do we need to worry here?

```
heroes.filter(function(hero) {
  nastyStateVariable = "Hey, I do something with side effects";
  return hero.kind === "robot";
}).map(function(hero) {
  return hero.name;
});
```

** it is because the return value is an expression, and in an expression never ever should happen anything else, just returning a value.*

SPREAD AND REST SYNTAX

Spread syntax – we take it apart!

from iterables → to multiple values

With spread syntax the iterables (string, Array, Map, Set, array-like objects) can be expanded to places where multiple values are expected.

add elements to an array →

```
> const robots = ["Wall-E", "EVE"];

  ["Captain", ...robots];
◂ ▶ (3) ["Captain", "Wall-E", "EVE"]
```

(shallow) copy an array →

```
> const copiedRobots = [...robots];

  copiedRobots;
◂ ▶ (2) ["Wall-E", "EVE"]
```

use as values when (multiple) parameters are expected →

```
> const copiedRobots2 = [].concat(...robots);

  copiedRobots2;
◂ ▶ (2) ["Wall-E", "EVE"]
```

a string is also an iterable →

```
> [..."Captain"];
◂ ▶ (7) ["C", "a", "p", "t", "a", "i", "n"]
```

Spread syntax – works for objects

from Objects → to an object literal

Spread syntax also can be used to break down objects into key-value pairs in object literals

this is a very powerful syntax, as we need to transform objects all the time →

```
> const kindObject = {
    kind: "human"
  };

  let hero = {
    name: "Captain",
    ...kindObject,
  }
  hero;
<· ▶ {name: "Captain", kind: "human"}
```

Spread syntax – objects, conditionally

Also, it can be used conditionally!

if it the condition is truthy, then it spreads and overrides the already defined properties

```
> const kindObject = {
    kind: "human"
  };

  let nameObject = {
    name: "Mary"
  };

  let hero = {
    name: "Captain",
    ...kindObject,
    ...nameObject && nameObject,
  }
  hero;
◂ ▶ {name: "Mary", kind: "human"}
```

```
> nameObject = undefined;

  hero = {
    name: "Captain",
    ...kindObject,
    ...nameObject && nameObject,
  }
  hero;
◂ ▶ {name: "Captain", kind: "human"}
```

if falsy, it does nothing

Spread syntax – funny cases

an array!

```
> const robots = ["Wall-E", "EVE"];
```

but arrays are objects as well

```
const hero = {
  ...robots
};

hero;
```

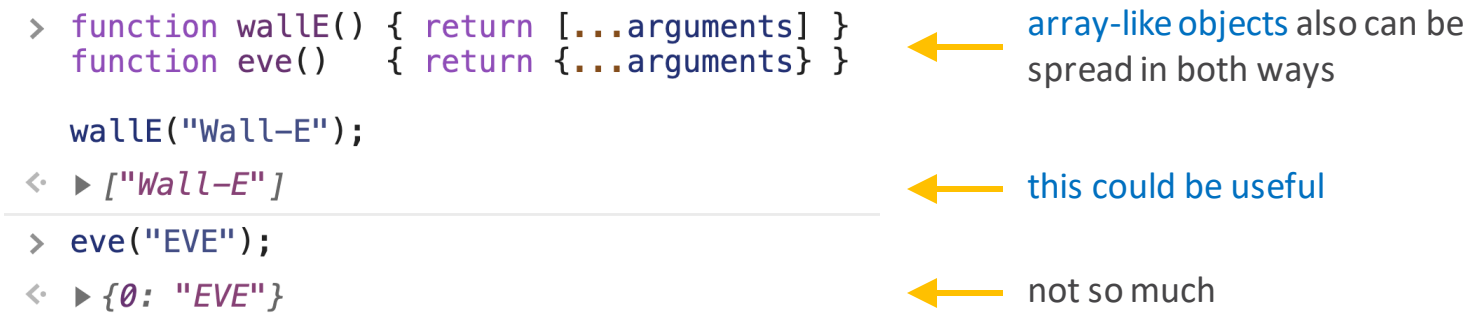
```
{0: "Wall-E", 1: "EVE"}
    0: "Wall-E"
    1: "EVE"
  ▶ __proto__: Object
```

so spread is working, but probably not in a way you'd intend to do

```
> ({..."EVE"})
{0: "E", 1: "V", 2: "E"}
    0: "E"
    1: "V"
    2: "E"
  ▶ __proto__: Object
```

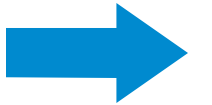
the same is true for strings (remember the String wrapper)

Spread syntax – funny cases II

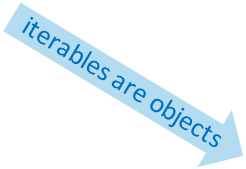


Spread syntax – summary

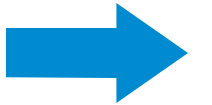
from any iterable* value:
array, string, Map, Set,
array-like objects (argument)



to multiple values



from Object



to Object literals

*basically, iterable values can be iterated with for…of

Rest parameter – assemble it again!

from individual values → to an array

The rest parameter is just the opposite to the spread syntax

While the spread syntax takes one thing apart, the rest assembles many parameters into an array. This array is a real array, not an array-like object like the arguments object.

things starts to get more interesting when we use many features together: arrow function + rest + IIFE

collecting all parameters into an array →

```
> ((...robots) => robots)("Wall-E", "EVE");
◂ ▶ (2) ["Wall-E", "EVE"]
```

exactly the same, just with a traditional function →

```
> function collectRobots(...robots) {
    return robots;
  }

  collectRobots("Wall-E", "EVE");
◂ ▶ (2) ["Wall-E", "EVE"]
```

Spread + Rest

It is fun when 2 completely different things look exactly the same…

```
> const heroes = ["Wall-E", "EVE"];

  ((...robots) => robots)(...heroes);

  ▶ (2) ["Wall-E", "EVE"]
```

a parameter list with rest parameters

a function call with spread syntax

but it could be confusing sometimes…

Spread + Rest

nope, we won't help now – it is your job to figure out what is going here...

```
> const robots = ["Wall-E", "EVE"];
  const humans = ["Captain", "Mary"];

  [...((...heroes) => heroes
    .map(hero => hero))(
      ...robots, ...((...heroes) => heroes
        .reduce((acc, hero) => [...acc, {
           kind: "human",
           name: hero
        }], []))(...humans))];
```

Rest with multiple parameters

Rest can be used with alongside other parameters

But only one rest parameter can exist, and it must be the last parameter.

”Wall-E” goes into the parameter *wallE*, and the rest go to the rest ;)

robots, the return value

nope, that won’t work

```
> const heroes = ["Wall-E", "EVE"];
  ((wallE, ...robots) => robots)(...heroes);
  ▶ ["EVE"]
> ((...robots, eve) => robots)(...heroes);
  Uncaught SyntaxError: Rest parameter must be last formal parameter
```

ARRAY AND OBJECT DESTRUCTURING

Destructuring assignment

from arrays and objects → to variables

The destructuring assignment makes it possible break down an array or object and assign its part into variables.

Destructuring could be positioned between the spread syntax and the rest parameters, as it breaks down a structure (spread), and assigns to a new variable (rest) at the same time.

destructuring is an assignment →

opps, wrong way →

a nice way to exchange variable values in one step →

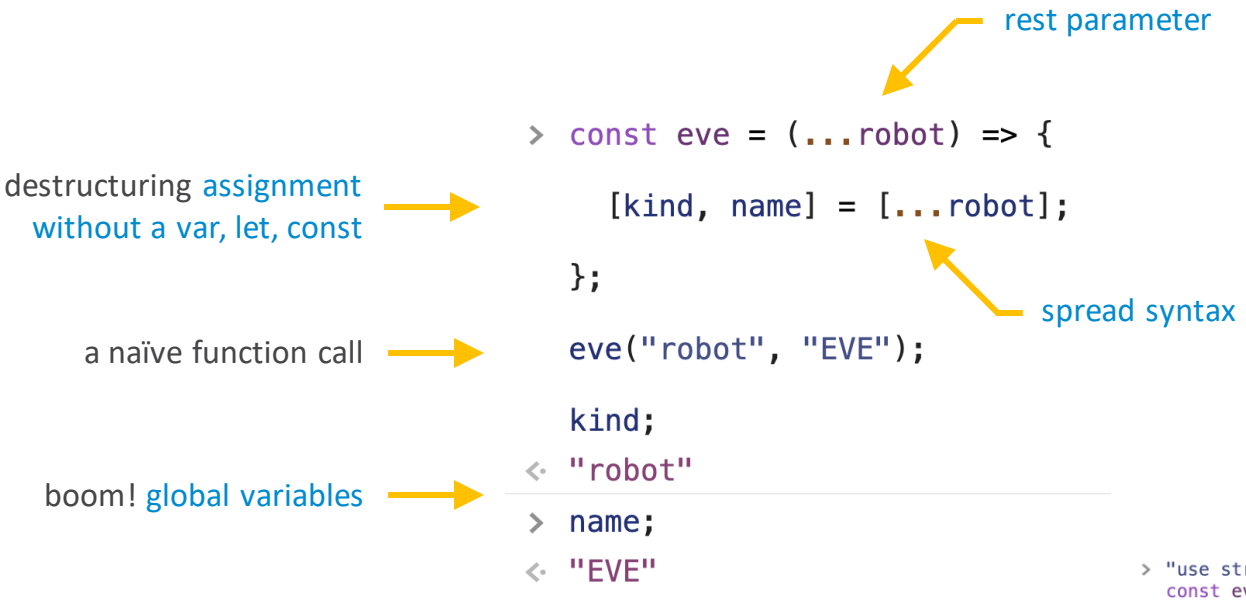
```
> const robots = ["Wall-E", "EVE"];

  let [eve, wallE] = robots;

  wallE;
<· "EVE"
> eve;
<· "Wall-E"
> [wallE, eve] = [eve, wallE];

  wallE;
<· "Wall-E"
> eve;
<· "EVE"
```

Destructuring assignment – a not so funny case




```
> const eve = (...robot) => {
    [kind, name] = [...robot];
  };
  eve("robot", "EVE");
  kind;
< "robot"
> name;
< "EVE"
```

```
> "use strict";
  const eve = (...robot) => [kind, name] = [...robot];

  eve("robot", "EVE");
  kind;
✕ ▶Uncaught ReferenceError: kind is not defined
      at eve (<anonymous>:2:28)
      at <anonymous>:4:1
```

Destructuring assignment – with rest element

destructuring to array

individual variables

and the rest go to the *rest*

the rest element must be the last
(like in rest parameters)

```
> const robots = ["Wall-E", "EVE", "Optimus Prime", "R2-D2", "Mikrobi"];
  let [wallE, eve, ...otherRobots] = robots;
  wallE;
"Wall-E"
> eve;
"EVE"
> otherRobots;
▶ (3) ["Optimus Prime", "R2-D2", "Mikrobi"]
> [wallE, ...otherRobots, eve] = robots;
✕ Uncaught SyntaxError: Rest element must be last element
```

rest element

Destructuring objects

Working with objects (as a data structure) is a major part of the development. Therefore, object destructuring used frequently to reduce the mass of the code.

```
> const robots = {
    wallE: "Wall-E",
    eve: "EVE"
  };

  const orgWallE = robots.wallE;
  const orgEve = robots.eve;

  const { wallE, eve } = robots;

  orgWallE;
< "Wall-E"
> wallE;
< "Wall-E"
```

object must be repeated and repeated again => *code noise*

traditional approach, explicit, but could be very verbose

destructuring

assigned variables

Destructuring objects – default values

properties can be chosen selectively,
(this is not the movie of R2-D2)
the order also does not matter, but the
variable name should be equal to a property

opps, something went wrong

in case of objects, parentheses are required if a var, let, const is missing

providing default value is possible

a missing value is still undefined

```
> const robots = {
    wallE: "Wall-E",
    r2d2: "R2-D2",
    eve: "EVE"
  };

  const { eve, wallE } = robots;

  wallE;
<· "Wall-E"
> eve;
<· "EVE"
> { mikrobi = "Mikrobi", optimusPrice } = robots;
✕ Uncaught SyntaxError: Unexpected token '='
> ({ mikrobi = "Mikrobi", optimusPrice } = robots);

  mikrobi;
<· "Mikrobi"
> optimusPrice;
<· undefined
```

Destructuring objects – rest properties

In case you wondered whether we have a 3rd kind of rest (after rest parameters, rest elements) - there you have: rest properties!

a (not so) complex object →

```
> const robots = {
    wallE: "Wall-E",
    eve: "EVE",
    r2d2: "R2-D2",
    mikrobi: "Mikrobi",
  };
```

picking up some →

```
  const { wallE, eve, ...otherRobots } = robots;

  otherRobots;
```

having the rest →

```
<· ▶ {r2d2: "R2-D2", mikrobi: "Mikrobi"}
```

Destructuring objects – new variable names

Restrictions about variable names can be lifted by providing new names. This could be necessary with special property names which are not valid JavaScript identifiers otherwise.

problem: *Wall-E* is not a valid identifier

different variable names than the properties

the new assigned variables

```
> const robots = {
    "Wall-E": "Wall-E",
    "EVE": "EVE"
  };

  const { "Wall-E": wallE, "EVE": eve } = robots;

  wallE;
< "Wall-E"
> eve;
< "EVE"
```

beware! these are not key - property pairs

Array and object destructuring – combined

We can combine the array and object restructuring – of course!

```
> const robots = [
    "EVE",
    {
      wallE: "Wall-E"
    }
  ];

  const [ eve, { wallE } ] = robots;

  wallE;
< "Wall-E"
```

the syntax is the same

we unpack a property from an object from an element of an array

it is pretty simple, right? wait for it...

Array and object destructuring – combined with default values

Can you follow this?

arrow function returning an array,
but what array? - you may ask…

… it is the robots array

this is not an array with objects,
but an *object destructuring* in
an *array destructuring*

missing *wallE* property

missing array element
(the object itself)

defaulting

destructuring

defaulting

```
> let eve;

  const setRobots = () => [
    eve,
    { wallE: earthRobot = "default of object deconstruction" } =
    { wallE: "default of array deconstruction" } ] = robots;

  let robots = [
    "EVE",
    {}
  ]
  setRobots();
  earthRobot;
<· "default of object deconstruction"

> let robots = [
    "EVE",
  ]
  setRobots();
  earthRobot;
<· "default of array deconstruction"
```

SHORTHAND PROPERTIES AND METHODS, COMPUTED PROPERTY NAMES

Object property shorthand

With object destructuring we could end up with a lots of variables, and there is a chance that we would like to structure a new object from them.

this is not that DRY →

no more duplication of identifiers →

```
> const wallE = "Wall-E";
  const eve = "EVE";

  const robots = {
    wallE: wallE,
    eve: eve
  };

  const newRobots = {
    wallE,
    eve
  };

  robots;
<· ▶ {wallE: "Wall-E", eve: "EVE"}
> newRobots;
<· ▶ {wallE: "Wall-E", eve: "EVE"}
```

Shorthand methods

The shorthand method definition also simplifies our object literals

Defining methods (with body) in object literals is often used in some cases, for example, when a 3rd party library's configuration requires some of its API methods to be overridden.

traditional →

less explicit, but shorter →

```
> const wallE = "Wall-E";
  const eve = () => "EVE";

  const longName = {
    wallE: function() {
      return "Waste Allocation Load Lifter: Earth-Class";
    },
    eve() {
      return "Extra-Terrestrial Vegetation Evaluator";
    }
  };

  longName.wallE();
<· "Waste Allocation Load Lifter: Earth-Class"
> longName.eve();
<· "Extra-Terrestrial Vegetation Evaluator"
```

Computed property names

Sometimes it is useful, when property names are not hard-wired values

traditional way

```
> const wallE = "Wall-E";
  const eve = () => "EVE";

  const names = {};
  names[wallE] = "Wall-E";
  names[eve()] = "EVE";
  names["R2D2".replace(/(R2)/, "$1-")] = "R2-D2";

  names;
◂ ▶ {Wall-E: "Wall-E", EVE: "EVE", R2-D2: "R2-D2"}
```

computed property names,
any expression could work here

```
> const longNames = {
    [ wallE ]: "Waste Allocation Load Lifter: Earth-Class",
    [ eve() ]: "Extra-Terrestrial Vegetation Evaluator",
    [ "R2D2".replace(/(R2)/, "$1-") ]: "Reel Two, Dialogue Two"
  };

  longNames;
◂ ▶ {Wall-E: "Waste Allocation Load Lifter: Earth-Class", EVE:
  "Extra-Terrestrial Vegetation Evaluator", R2-D2: "Reel Two,
  Dialogue Two"}
```

You may realize now that with the excessive usage of these JavaScript features the readability can be out of hand really quickly

Is it required to understand and use these complex structures?

Well, *I have good news*: it is pretty rare when *spread, rest, destructuring with defaults, etc.* are used in one statement or expression.

Also, you can use these, but you don't have to. It is the matter of the code conventions of the project whether these are used at all and in what way.

Generally speaking, however, you should expect that you will meet all these in project code.

That being said, the code you need to work with will be very complex, by nature. You get used to it with time – no worries!

edu_hu@epam.com