ECMA-262

EcmaScript 6

Üstün Özgür
@ustunozgur
ustunozgur.com

sellercrowd

Hello there, my name is Ustun and the topic of today's talk will be EcmaScript 6, the next specification for the JavaScript language. Before I move on to the presentation, let me introduce myself. I'm a software engineer from Turkey, visiting Barcelona for the month. Currently I'm leading the development of a social networking site called SellerCrowd.

History

- JavaScript: Implementation
- EcmaScript: Specification
- Netscape JavaScript ve Microsoft JScript, ActionScript
- ECMA standard organization project no. 262
- TC-39: Technical Committee 39

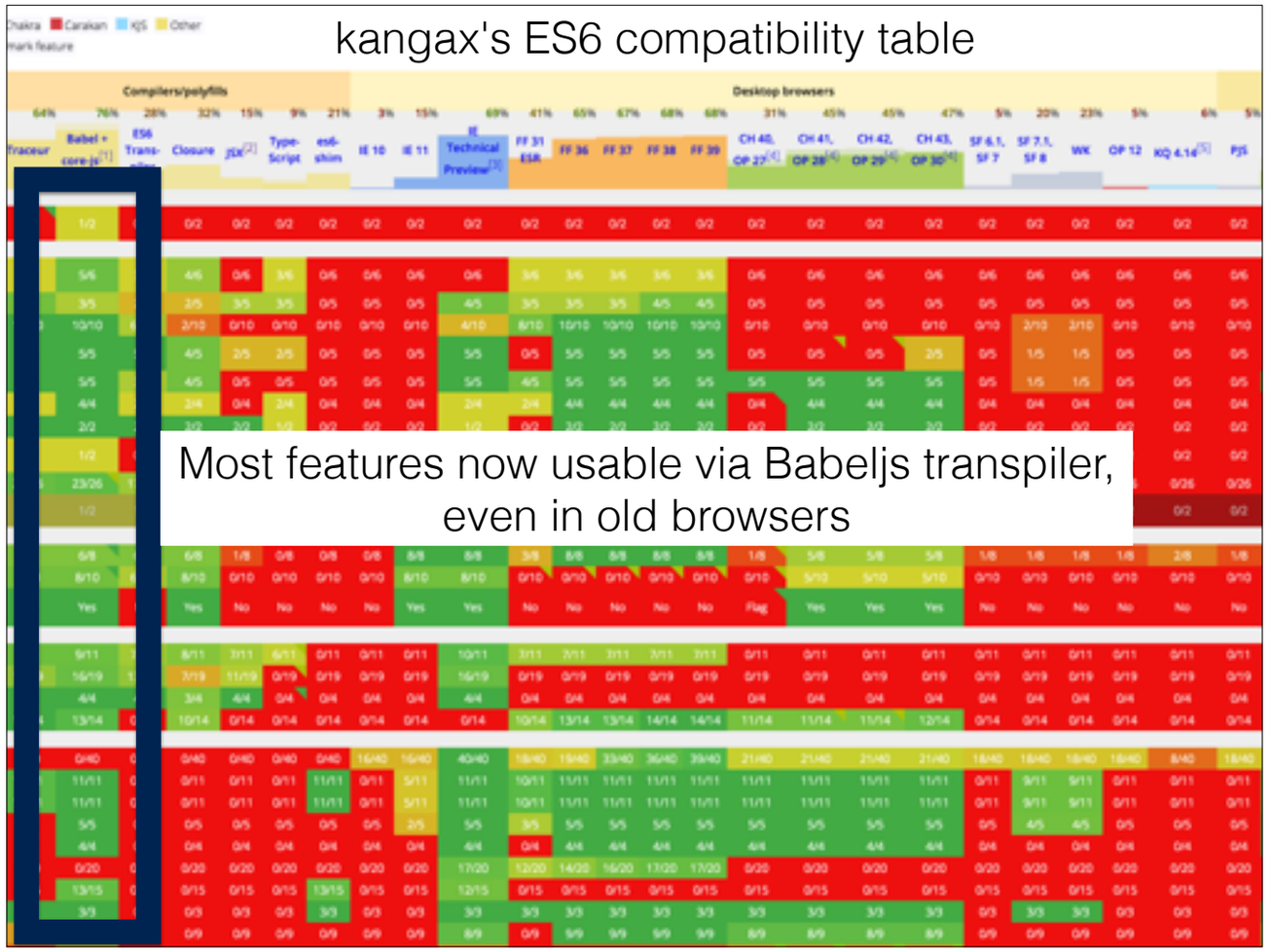
Let's first give a brief overview of the history of Ecmascript. First we have to distinguish between JavaScript and Ecmascript. When JavaScript was first invented at Netscape, it didn't have a specification. Later on, Microsoft implemented its own version called JScript and then the need for an implementation independent specification came out. Thus Ecmascript was born, part of the Ecma standards organization, with project code 269, led by the tech committee 39.

History

- ES 3: 1999
- ES 4: Abandoned
- ES 5 : 2009 and 5.1: 2011
  - 258 pages
- ES 6: Draft ready
  - 657 pages
  - ETA June 2015
  - Harmony
- ES 7: Already in progress

ES3 was released in 1999. Then came the development of ES4, which was quite ambitious. Unfortunately due to problems within the committee this release was abandoned, and some members of the committee created ES5, which was based on version 3. ES5, which most modern browsers implement was released in 2009 and 5.1 was released in 2011.

Then work started on ES6, which is almost ready, currently in its final draft form. It is expected that the final version will be released this June. This also marks the most ambitious version ever. For comparison, ES6 draft is 657 pages long, whereas ES5 was only 258 pages. It also has the codename "harmony", which represents the harmony within the technical committee.



Even though it is not released yet, browsers have already started implementing ES6 features. However, if we look at this compatibility table with current browsers, where green indicates that the feature is supported, we see that most browsers still do not natively support most of the features. But if we look closely to the leftmost column which represent the Babel transpiler, we see that most features are already available via this project. This means we can use ES6 features in our code now, and use Babel transpiler to get the transpiled code that runs in any browser.

Overview

- Let ve Const keywords vs Var
- Changes in functions
- Changes in objects and destructuring
- Classes
- Template strings
- Promises

Let me first give an overview of the new features in ES6. In fact, there are so many features that I had to select only the most important ones. We can summarizes the changes as follows: let and const keywords instead of var, changes in functions, changes in objects, classes, template strings and promises. These are the main features I'll be talking about today.

Let and Const

- JS, looks like C or Java due to curly brace blocks
  - but no block scope
- Vars are function scoped
- Let ve Const block scoped
- Causes confusion for beginners

First, let's take a look at let and const. JavaScript looks like C or Java due to curly brace blocks, but unlike them, it has no block scope. A var keyword introduces a variable that is function scoped. This introduces lots of confusion, especially among beginners. The let keyword solves this problem by introducing block scoped variables.

Var and Let

Hoisting

```
function foo () {
  var i;
  console.log(i); undefined

  for (var i = 0; i < 10; i++) {
     console.log(i);
  }

  console.log(i); 10
}
```

Let's first see an example. In this example, we have a for-loop where we define the variable i. However, due to hoisting, this declaration moves to the top of the function and the variable i is defined in the whole function. Therefore, this snippet will first print undefined and finally 10.

Var ve Let

```
function foo () {

  console.log(i); // ERROR

  for (let i = 0; i < 10; i++) {
      console.log(i); // i defined only in this block
  }

  console.log(i); // ERROR
}
```

Let simply solves this issue so that the variable i is only defined within the block. It is an error to refer to i if it is not defined there.

This example might look too simplistic and it is, and I'll show a real-life demo where this scoping is important if we have time at the end.

Const

```
const PI = 3.14;

PI = 3; // ERROR
```

Better to have values than variables for less bugs

DEMO
let_const.js

Next, const. Const, as its name suggests is simply a constant value that cannot be changed. You will get a transpilation error if you try to change a const value. It is better if we have more values rather than variables in our codebase, doing so will decrease the number of bugs we encounter. So my advice is to stick to const usually and then use let if needed.

Changes in Functions

- Default parameters **function foo(**name="Ustun"**)**
- Rest parameters **function foo(name,** ...rest**)**
- Destructured parameters **function foo(**{name, surname}**)**

Now, let's continue with the second section, where we discuss the changes made to functions. While defining functions, three major changes have been introduced. First we can define default values for parameters. Two, there is new syntax for getting the rest of the arguments as an array, shown here with 3 dots or ellipses. Three, we can do destructuring on the function parameters.

Default Params

```
function hello(name="Ustun", greeting="Hello") {
    console.log(greeting + " " + name);
}

hello();

hello("Ahmet");

hello("Mehmet", "Hola");
```

Not keyword params like in Python!

~~hello(greeting="Hello", name="Ozgur")~~

DEMO
functions_default_params.js

Let's look at an example for default parameters. Here I define a function called hello that takes name and greeting as arguments. We can provide default values for these two arguments here, and if they are not supplied the default values will be used. Note that these are not keyword arguments, so you cannot swap the place of arguments as you can in Python.

Rest params

```
function sum(firstValue, ...rest) {

var total = firstValue;

      for (var i = 0; i < rest.length; i++) {

        total += rest[i];

    }

     return total;

}
```

sum(15, 1, 2, 3);

rest = [1, 2, 3]

DEMO
functions_rest_params.js

Now, rest parameters. In JavaScript, the number of arguments a function takes is not fixed. You can pass however many arguments to a function and those will be collected in an array-like structure called arguments. Note however that arguments is not a real array, so
some methods like slice are missing from it. If we want to take not the whole argument list as an array, but only the ones after one or two values, we can now use the rest parameter syntax. In this example for example, the first argument will be assigned to initial value and the remaining arguments will be collected as an array in the variable rest. So, rest arguments is for when the function is being defined.

Spread operator

- Math.max(1, 2, 3); 3
- Math.max([1, 2, 3]); // NaN
- var a = [1, 2, 3]; Math.max(a); // NaN
- Math.max.apply(Math, a); // 3
- Math.max(...a); // 3

You can also use ellipses while calling a function. In that case, the ellipses will take an array value and spread it over as if its contents were passed as individual arguments. Let's see an example here. Math.max function is variadic, that is it accepts arguments an indefinite number of arguments one by one. If we try to pass an array to it to see the max value in the array, we get the value NaN. This is because it does not accept an array as argument. Previously the way to overcome this limitation was to use apply method, where we supply the argument list the function will get. However this is quite bulky, because with apply, we also have to pass the parameter that the keyword this will be assigned to. So, here we have to mention Math twice. With ES6, we can use an ellipses which will spread an array to the argument list so that Math.max works with an array.

Parameter Destructuring

```
function hello(name, options) {
    var greeting;
    var lang = options.lang;
    if (lang == "en") greeting = "Hello";
    if (lang == "es") greeting = "Hola";
    return greeting + " " + name;
}
```

```
function hello(name, {lang}) {
var greeting;
    if (lang == "en") greeting = "Hello";
    if (lang == "es") greeting = "Hola";
    return greeting + " " + name;
}
```

DEMO
functions_param_destructuring.js

Final improvement to function syntax is parameter structuring. Let's say you have a function that accepts an object as an argument. But let's say you are only interested in only a few fields of this array. What you would do in this case is to access that field in the argument. Now, however, you can indicate that you are only interested in specific fields in an incoming object parameter. For example, here, with the curly braces next to lang, we indicate that we are only interested in the lang field of the object. This is called parameter destructuring, because the incoming object is destructured and only certain parts are used.

```
function setCookie(name, value, options) {

    options = options || {};

    var secure = options.secure,
        path = options.path,
        domain = options.domain,
        expires = options.expires;

    // ...
}

setCookie("type", "js", {
    secure: true,
    expires: 60000
});


function setCookie(name, value, { secure, path, domain, expires }) {

    // ...
}
```

Parameter destructuring is usually most useful where you have an options object in the signature of the function. Here we see a setCookie example that takes an option object as an argument. In the second version, we do the destructuring automatically when we define the function. This also aids code readability because someone reading this code later on will see the fields we are interested in just by reading the function argument list, without referring to the function body.

Arrow functions

```
var bar = (a, b) => a + b;
```

```
var foo = function (a,b) {

 return a + b; }

 foo(1,2) ; // 3

 bar(1,2); // 3
```

ES6 also introduces a new function type called arrow functions, which uses a fat arrow, that is an equal sign followed by a greater than sign. Here we have a function that takes as input a and b and returns a plus b. No need for function or return keywords, or curly braces. This is almost equivalent to this verbose form we already known.

- var nums = [1, 2, 3, 4];
- nums.filter(x => x % 2 === 1)
- nums.reduce((a,b) => a * b)

DEMO
arrow_functions.js

Let's see another example. Here we have a list of numbers and we want to find out the odd numbers. We simply pass an arrow function that tests for oddness. Similarly, in the last example here, we are passing a function that multiplies its inputs. The output of this reduce operation will be the multiplication of all values in this array, so 24 in this example.

Arrow functions and this keyword

```
var x = {
    name: "Ustun",
    hello: function () {
        var that = this;
        var helper = function () {
            console.log("Name ", this.name);
        };
        helper();
    }
};
x.hello()
```

that

Remember that I said arrow functions are almost equivalent to the old function syntax. They are not the same though and the main reason is the following. The keyword this in an arrow function is lexically scoped. That is, this will refer to whatever this refers to where this function resides. To see an example where this causes an issue, consider this example. In an object, I define a method, which defines a helper function. Now, when that helper function is called, the keyword this will not refer to the object, since it is not bound to the object. A function called on its own will have the global window as the this value, since it is unbound. To fix this issue, we can use the following trick. We first defined a variable called that in the outer scope, and refer to that, not this. When we do that, that will now always refer to the object where it is defined in.

```
var x = {
    name: "Ustun",
    hello: function () {

        var helper = function () {
            console.log("Name ", this.name);
        }.bind(this);
        helper();
    }
};
x.hello()
```

Another solution to this problem is to use the bind method so that this refers to whatever we pass to bind. Here we define the function and then bind the this keyword.

this refers to the value where function is defined, not called
(lexical scope vs dynamic scope)

```
var x = {
    name: "Ustun",
    hello: function () {
        var helper = () => {
            console.log("Name ", this.name);
        };
        helper();
    }
};
x.hello()
```

DEMO
this.js

What arrow functions do is to solve this common situation, by binding this automatically. Here, we define the same helper function, we refer to this, which will refer to the this variable where this function was defined in.

Changes in Objects

Destructuring

```
var ustun = {name: "Ustun", lastname: "Ozgur"}

var name = ustun.name;

var lastname = ustun.lastname;

var {name, lastname} = {name: "Ustun", lastname: "Ozgur"}


var {name: nombre} = ustun;
```

Now, we will go over the changes in the object syntax. First destructuring. We already saw destructuring from objects in the function context, but we can also use destructuring when defining variables. Here on the right we have an object and we pluck out name and lastname fields from the object to their own variables. We can also rename the variable while destructuring. In the last example, we will destructure the name field and assign it to a variable called nombre.

Destructuring

- Array destructuring
- var [a,b] = [1,2];
- var [a,b] = [b,a]; // Swap
- Deep destructuring possible

We can also destructure arrays, so that the first element on the right will be assigned to the first on the left and so on. Note that this is doing a match on the pattern, but it is loose. If the array on the right is longer, it won't fail, it will simply ignore the remaining values. Similarly, if the array on the left is longer, the remaining values will be set to undefined.

We also have to note that deep destructuring, that is, destructuring at a few levels is possible. We will also see an example for this if time permits.

Shorthand for Object Creation

age = 30; name = "Ustun"; location = "Turkey";

ustun = {name: name, age: age, location: location};

age = 45; name = "Jose"; location = "Barcelona";

ahmet = **{name, age, location}**;

DEMO
objects.js

As the reverse operation for destructuring, ES6 introduces a shorthand for creating objects. For example, let's say we have a few variables name, surname and id, which we will use as the fields of an object. We used to be able to define the object by repeating the field names, that is {name: name, surname:surname}. ES6 solves this common usecase by allowing us to type just the variable names to create the object.

Shorthand for Object Creation

```
var ustun = {
  name: "Ustun",
  sayName: function () {
    console.log("I'm " + this.name);
  }
}
```

```
var ustun = {
  name: "Ustun",
  sayName() {
    console.log("I'm " + this.name);
  }
}
```

Another convenient feature is that while defining functions on the objects, we no longer have to type out function keyword and the semicolon, if the fieldname has parens next to it, it is a function.

Template Strings

```
name = "Ustun", age = 30;

console.log("I'm " + name + ".Yasim " + age);

console.log(`I'm ${name}. My age ${age}`);

console.log(`This spans

multiple lines`);
```

Another important change in ES6 is template strings. Template strings basically allow you to define a template string, where the contents of the $ expressions can refer to JS values. So, instead of string concetenation, you can refer to them inside the template string. You can also have multiline strings this way, which was not possible in JavaScript unless you put a slash at the end of each line which was not recommended.

Tagged Template Strings

- functionName`Hello ${name}`;
- safe`Hello ${name}`;
- uppercase`Hello ${name}`;
- var safe = function (literals, ...variables) { ...}
- var uppercase = function (literals, ...variables) {...}

The most important feature of template strings however is tagged template strings. Before the string, you can refer to a function which will process the tokens of the string before outputting the final string. For example, you can define a function called safe, that will escape and sanitize the tokens against XSS attacks.

Class Keyword

- class and extends
- constructor
- transpiled to prototypes

ES6 also introduces the class and extends keywords, so that we can now define classes similar to Java. Behind the scenes, this still uses prototypal inheritance. The syntax is straightforward. If you have to refer to a method in the super class, you use the super keyword.

Classes

```
class Human {

    constructor(name, age) {

        this.name = name;

        this.age = age;

        this.party = null;

    }
```

```
class Ogrenci extends Human {

  constructor(name, age, school) {

        super(name, age);

        this.school = school;

  }
```

Other features

- Modules
- Promises
- Generators

There are a few other important features in ES6 that I haven't had the time to talk about. The most important ones are modules, promises and generators.

Babel.js

- Transpiler
- babel source.js > destination.js
- babel --experimental source.js
- require('babel/polyfill')

Finally, I wanted to mention the transpiler I used to run these code examples again. It is called babel js. You simply transform your ES6 code through babel and it will produce code compatible with browsers and node.js.

More Info

- Wiki: http://wiki.ecmascript.org/
- Understanding ECMAScript 6
  - https://leanpub.com/understandinges6/
- Taslaklar: http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts
- babeljs.io
- https://github.com/lukehoban/es6features
- http://kangax.github.io/compat-table/es6/

Thank you!

- @ustunozgur
- https://github.com/ustun/ecmascript6-presentation
- ustun@ustunozgur.com
- **NEXT WEEK: React.js talk at BarcelonaJS**

Extras

Example

```
var links = document.getElementsByTagName('a')

for (var i = 0, len = links.length; i < len; i++){

   links[i].namedEventListener('click', function(e){

        alert('You clicked on link ' + i)

    }, false)

}
```

DEMO

var_problem.html

Solution

Enclose the var in a function

```
var links = document.getElementsByTagName('a')
for (var i = 0, len = links.length; i < len; i++){
  (function (j) {
     links[j].namedEventListener('click', function(e){
         alert('You clicked on link ' + j)
      }, false)
  })(i);
}
```

DEMO

var_cozum.html

Solution: Let

```
var links = document.getElementsByTagName('a')

for (let i = 0, len = links.length; i < len; i++){

   links[i].namedEventListener('click', function(e){

        alert('You clicked on link ' + i)

    }, false)

}
```

DEMO

let_cozum.html

Computed Properties

```
var fieldName = "firstName";

var a = {

 fieldName: "ustun";

 }
```

```
a.fieldName == "ustun"

a.firstName ? undefined

a[fieldName] = "ustun";

a.firstName == "ustun"
```

```
var a = {

  [fieldName]: "ustun";

}
```

List Comprehensions

- var x = [for (i of [0, 1, 2, 3]) i * i];
- [0, 1, 4, 9]

- var y = [for (i of [0, 1, 2, 3]) if (i % 2 === 0) i * i * i];
- [0, 8]

Promises

```
student = findStudent(123)

className = findClass(student)

school = findSchool(className)
```

The final feature I'll talk about is promises. Let's say that we have the following code, which first finds a student, then based on the student, finds his classroom and finally his school. Now assume that each function takes a few seconds. This means that we can no longer write this as shown here, otherwise it will block the whole page since JavaScript is single threaded. What is usually done is to alter these functions so that they are async and take a callback as another parameter.

Callbacks & Pyramid of Doom

```
student = findStudent(123, function (student) {

    findClass(student, function (className) {

        findSchool(className, function (school) {

            console.log(school);

        }}}
```

So we modify the codebase as follows. Each function takes a callback that calls the next function in the program. This solves the issue, at the code of too much nesting. This problem is called the pyramid of doom. Promises solve this issue by linearizing the flow, so that each function can take the output of the further promise. When written this way, the code block is again readable. Promises can actually be implemented at the library level, so you might ask yourself why this needed to be included in the standard. The reason is that the API's in the browser can use promises if they are defined in the language itself. Also, it provides convenience to the programmer, so that he does not have to include a library.

Pyramid of Doom Sol'n

```
findStudent(123)
    .then(findClass)
    .then(findSchool)
    .then(function (school) {console.log(school);}
    .catch(function () { console.log("error")})
```

To convert a function to a promise, we return a new Promise function that accepts two methods, resolve and reject, and we call the resolve at the end of our original code. Other nice features of Promises are the static methods all and race. All takes a list of promises and returns another promise that will be resolved when all of the promises in it are resolved. Race takes a list of promises, and returns another promise that will be resolved as soon as one of the promises have been resolved.

Promise

- new Promise(resolve, reject)
- Promise.all([promise1, promise2]).then
- Promise.race([promise1, promise2]).then

DEMO

Modules

- import myfunc from mylib;
- export myfunc;