ES6 (ES2015) +

GOCODE

What is ES6 (ES2015) ?

As of June 2015, the largest addition to the JavaScript language was finalized. The official name is ECMAScript 2015, sometimes referred to as "ES6", or now more commonly known as "ES2105". It is the culmination of years of work and features.

Moving forward, there will be ECMAScript 2016, which will likely be referred to as "ES7" or "ES2016". The plan is to have incremental yearly releases.

"var" function Scoping

a variable defined exclusively within the function cannot be accessed from outside the function or within other functions

```
function exampleFunction() {
    var x = "declared inside function";  // x can only be used in exampleFunction
    console.log("Inside function");
    console.log(x);
}

console.log(x);  // Causes error
```

"var" global Scoping

However, the following code is valid due to the variable being declared outside the function, making it global:

```
var x = "declared outside function";


exampleFunction();


function exampleFunction() {
    console.log("Inside function");
    console.log(x);
}


console.log("Outside function");
console.log(x);
```

"var" has no block scope

var variables are still visible after blocks

```
if (true) {
  var test = true;
}

alert(test); // true, the variable lives after if



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

alert(i); // 10, "i" is visible after loop, it's a global variable
```

"var" hoisting

var variables **declarations** are hoisted (raised) to the top of the function
(Declarations are hoisted, but assignments are not.)

```
function sayHi() {
  phrase = "Hello"; // assignment

  if (false) {
    var phrase; // declaration
  }

  alert(phrase);
}
```

let & const Variables

let & **const** are **scoped-variables.** When you define a variable with the let keyword it will create a new variable only within the { } or block.

```
{
  var user = "David";
}
console.log(user); // David
```

```
{
  let user = "David";
}
console.log(user); // Uncaught ReferenceError: user is not defined
```

so use let instead of var

in a for loop:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
console.log(i); // Uncaught ReferenceError: i is not defined
```

use const for read-only variables

When you use const for a primitive variable (number, string) - you can't change it after the firsr declaration & assignment. To know it's a const variable, we should give it an UPPER-CASE name.

```
const PERSON = 'David';
PERSON = 'Moshe'; // Uncaught TypeError: Assignment to constant variable.
console.log(PERSON);
```

```
const MILLISECONDS_UNTIL_ALERT = 3000;
setTimeout(function() {
  alert('ALERT!');
}, MILLISECONDS_UNTIL_ALERT)
```

const on object still let you to reassign values

Because it's still the same object and you just change the value inside it…

To make the object be a readonly object, use **Object.freeze**

```
const person = {
  name: 'David'
};
person.name = 'Moshe';

console.log(person); // {name: 'Moshe'}
```

```
const person = Object.freeze({
  name: 'David'
});
person.name = 'Moshe';

console.log(person); // {name: 'David'}
```

Template literals (template strings)

Template literals are enclosed by the back-tick (` `) (grave accent) character instead of double or single quotes. Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (${expression}) and can be multi-line

```
let a = 5;
let b = 10;
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
```

```
let a = 5;
let b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
```

Tagged templates

Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.

```
let person = 'Mike';
let age = 28;
function myTag(strings, personExp, ageExp) {
  let str0 = strings[0]; // "That "
  let str1 = strings[1]; // " is a "
  let ageStr;
  if (ageExp > 99){
    ageStr = 'centenarian';
  } else {
    ageStr = 'youngster';
  }
  // We can even return a string built using a template literal
  return `${str0}${personExp}${str1}${ageStr}`;
}
```

```
let output = myTag`That ${ person } is a ${ age }`;
console.log(output);
// That Mike is a youngster
```

Arrow functions

Arrow functions are a new syntax for creating functions in ES2015. This does not replace the function() {} syntax that we know and love, but we will use it as a shorter function syntax.

```
const add = (a, b) => {
  return a + b;
};
add(2, 3); // 5
```

```
const add = (a, b) => a + b;
```

Arrow functions syntax

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter name:
(singleParam) => { statements }
singleParam => { statements }

// The parameter list for a function with no parameters should be written with a pair of
parentheses.
() => { statements }
```

Arrow functions syntax example

```
let elements = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];
```

```
elements.map(function(element) {
  return element.length;
}); // this statement returns the array: [8, 6, 7, 9]

// The regular function above can be written as the arrow function below
elements.map((element) => {
  return element.length;
}); // [8, 6, 7, 9]

// When there is only one parameter, we can remove the surrounding parenthesies:
elements.map(element => {
  return element.length;
}); // [8, 6, 7, 9]

elements.map(element => element.length); // [8, 6, 7, 9]
```

Arrow functions don't have their context!

```
let person = {
   age: 20,
   firstName: 'Avi',
   lastName: 'Levi',
   fullName: function () {return this.firstName + ' ' + this.lastName}
}
console.log('FullName: , person.fullName()); // FullName: Avi Levi
```

```
let person = {
   age: 20,
   firstName: 'Avi',
   lastName: 'Levi',
   fullName: () => this.firstName + ' ' + this.lastName
 }

console.log('FullName: , person.fullName());  // FullName: undefined undefined
```

"**this**" refers to **window** in this case...

Use arrow function when you don't want to use context

```
const person = {
  firstName: 'Ryan',
  hobbies: ['Robots', 'Games', 'Internet'],
  showHobbies: function() {
    this.hobbies.forEach(function(hobby) {
      console.log(`${this.firstName} likes ${hobby}`);
    });
  }
};
person.showHobbies();
```

OUTPUT:

```
undefined likes Robots
undefined likes Games
undefined likes Internet
```

```
const person = {
  firstName: 'Ryan',
  hobbies: ['Robots', 'Games', 'Internet'],
  showHobbies: function() {
    this.hobbies.forEach(hobby => {
      console.log(`${this.firstName} likes ${hobby}`);
    });
  }
};
person.showHobbies();
```

OUTPUT:

```
Ryan likes Robots
Ryan likes Games
Ryan likes Internet
```

Another example with setTimeout context..

```
// ES5

var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(function() {
      console.log(this.id);
    }, 1000);
  }
};
// undefined! We need to use bind(this)...

var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(function() {
      console.log(this.id);
    }.bind(this), 1000);
  }
};
```

```
// ES6 - Arrow Function

var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(() => {
      console.log(this.id);
    }, 1000);
  }

};
```

So when we should not use Arrow Functions?

```
// On Object Methods:


var cat = {
  lives: 9,
  jumps: () => {
    console.log('jumps', this.lives);
    this.lives--;
  }
}

// this.lives on cat.jumps() is NaN
```

```
// Using anonymous function methods syntax
var cat = {
  lives: 9,
  jumps() {
    console.log('jumps', this.lives);
    this.lives--;
  }
}
// this.lives on cat.jumps() is 9. Similar to regular syntax:

var cat = {
  lives: 9,
  jumps: function() {
    console.log('jumps', this.lives);
    this.lives--;
  }
}
```

So when we should not use Arrow Functions?

```
// Callback functions with dynamic context

var button = document.getElementById('press');
button.addEventListener('click', () => {
  this.classList.toggle('on');
});
```

If we click the button, we would get a TypeError. It is because this is not bound to the button, but instead bound to its parent scope.

```
// So use anonymous function:

var button = document.getElementById('press');
button.addEventListener('click', function() {
  this.classList.toggle('on');
});
```

Spread operator

Spread syntax allows an iterable such as an array expression or string to be expanded in places

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

const numbers = [1, 2, 3];

console.log(sum(...numbers));
// expected output: 6

console.log(sum.apply(null, numbers));
// expected output: 6
```

Spread operator - syntax

For function calls:

```
myFunction(...iterableObj);
```

For array literals or strings:

```
[...Object.values(iterableObj), '4', 'five', 6];
```

For object literals (new in ECMAScript 2018):

```
let objClone = { ...obj };
```

Spread in array literals

Use array values to initialize another array:

```
let parts = ['shoulders', 'knees'];
let lyrics = ['head', ...parts, 'and', 'toes'];
// ["head", "shoulders", "knees", "and", "toes"]
```

Copy an array:

```
let arr = [1, 2, 3];
let arr2 = [...arr]; // like arr.slice()
arr2.push(4);

// arr2 becomes [1, 2, 3, 4]
// arr remains unaffected
```

Spread syntax effectively goes one level deep while copying an array!

```
let a = [[1], [2], [3]];
let b = [...a];
b[0][0] = '';
console.log('a:', a)
// Now array a is affected as well!
// a: [[""], [2], [3]]
```

Spread operator to concat arrays

This is how we do it with concat function:

```
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
// Append all items from arr2 onto arr1
arr1 = arr1.concat(arr2);
// arr1: [0, 1, 2, 3, 4, 5]
```

With spread syntax:

```
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
arr1 = [...arr1, ...arr2];
// arr1: [0, 1, 2, 3, 4, 5]
```

Spread operator to do unshift (add to the beginning)

This is how we do it with unshift function:

```
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
// Prepend all items from arr2 onto arr1
Array.prototype.unshift.apply(arr1, arr2)
// arr1 is now [3, 4, 5, 0, 1, 2]
```

With spread syntax:

```
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
arr1 = [...arr2, ...arr1]; // arr1 is now [3, 4, 5, 0, 1, 2]
```

Spread in object literals

The Rest/Spread Properties for ECMAScript proposal (stage 4) adds spread properties to object literals. It copies own enumerable properties from a provided object onto a new object.

```
let obj1 = { foo: 'bar', x: 42 };
let obj2 = { foo: 'baz', y: 13 };

let clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }

let mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

Rest parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

```
function sum(...theArgs) {
  return theArgs.reduce((previous, current) => {
    return previous + current;
  });
}

console.log(sum(1, 2, 3));
// expected output: 6

console.log(sum(1, 2, 3, 4));
// expected output: 10
```

Rest parameters

A function's last parameter can be prefixed with ... which will cause all remaining (user supplied) arguments to be placed within a "standard" javascript array. Only the last parameter can be a "rest parameter".

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("one", "two", "three", "four", "five", "six");

// Console Output:
// a, one
// b, two
// manyMoreArgs, [three, four, five, six]
```

Destructuring assignment - syntax

```
let a, b, rest;
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]
```

```
({ a, b } = { a: 10, b: 20 });
console.log(a); // 10
console.log(b); // 20


// Stage 4(finished) proposal
({a, b, ...rest} = {a: 10, b: 20, c: 30, d:
40});
console.log(a); // 10
console.log(b); // 20
console.log(rest); // {c: 30, d: 40}
```

Notes: The parentheses (...) around the assignment statement are required when using object literal destructuring assignment without a declaration.

{a, b} = {a: 1, b: 2} is not valid stand-alone syntax, as the {a, b} on the left-hand side is considered a block and not an object literal.

Destructuring assignment - with new vars names

```
let o = {p: 42, q: true};
let {p: foo, q: bar} = o;

console.log(foo); // 42
console.log(bar); // true
```

With default values:

```
let {a = 10, b = 5} = {a: 3};

console.log(a); // 3
console.log(b); // 5
```

Assigning to new variables names and providing default values

```
var {a: aa = 10, b: bb = 5} = {a: 3};

console.log(aa); // 3
console.log(bb); // 5
```

Default function parameters

Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

Before ES2015:

```
function multiply(a, b) {
  b = (typeof b !== 'undefined') ?  b : 1;
  return a * b;
}

multiply(5, 2); // 10
multiply(5);    // 5
```

After ES2015:

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5, 2); // 10
multiply(5);    // 5
```

More Info

1. https://javascript.info
2. https://eloquentjavascript.net
3.

Template strings exercise

1. Write a function that gets number of Dollars and Dollar to ILS currency rate, (for example - func(30, 3.61)) and return the following template string: You have 30 dollars that are equal to 108.3 shekels
2. https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/es6/create-strings-using-template-literals
3.