ECMAScript 6

ECMAScript (ES) is a scripting language specification created to standardize JavaScript.

The Sixth Edition, initially known as **ECMAScript 6** (ES6) and later renamed to **ECMAScript 2015**, adds significant new syntax for writing complex applications, including classes and modules, iterators and for/of loops, generators, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, number and math enhancements, reflection, and proxies.

In other words, ES6 is a superset of JavaScript (ES5). The reason that ES6 became so popular is that it introduced new conventions and OOP concepts such as classes.

In this module, we cover the most important additions to **ES6**.
So, let's jump right in!

JavaScript and ES6 are different technologies for different purposes.

`False`

`True`

var & let

In ES6 we have three ways of declaring variables:

```JS
var a = 10;
const b = 'hello';
let c = true;
```

The type of declaration used depends on the necessary **scope**. **Scope** is the fundamental concept in all programming languages that defines the visibility of a variable.

var & **let**

Unlike the **var** keyword, which defines a variable globally, or locally to an entire function regardless of block scope, **let** allows you to declare variables that are limited in scope to the block, statement, or expression in which they are used.

For example:

CODE PLAYGROUND JS

```
if (true) {
  let name = 'Jack';
}
alert(name); //generates an error

Tap to edit |
```

In this case, the **name** variable is accessible only in the scope of the **if** statement because it was declared as **let**.

To demonstrate the difference in scope between **var** and **let**, consider this example:

CODE PLAYGROUND JS

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

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

Tap to edit |

One of the best uses for **let** is in loops:

CODE PLAYGROUND **JS**

```
for (let i = 0; i < 3; i++) {
  document.write(i);
}

Tap to edit |
```

Here, the **i** variable is accessible only within the scope of the **for** loop, where it is needed.

let is not subject to **Variable Hoisting**, which means that **let** declarations do not move to the top of the current execution context.

What is the output of this code?

```
function letItBe() {
  let v = 2;
  if (true) {
    let v = 4;
    console.log(v);
  }
  console.log(v);
}
letItBe();
```

4 4

2 4

2 2

4 2

CHECK

const

const variables have the same scope as variables declared using **let**. The difference is that const variables are **immutable** - they are not allowed to be reassigned.
For example, the following generates an exception:

CODE PLAYGROUND **JS**

```
const a = 'Hello';
a = 'Bye';

Tap to edit |
```

const is not subject to **Variable Hoisting** too, which means that **const** declarations do not move to the top of the current execution context.
Also note that ES6 code will run only in browsers that support it. Older devices and browsers that do not support ES6 will return a syntax error.

Fill in the blanks to make a constant named total and the variable i that is only accessible inside the loop.

```
  total = 100;
let sum = 0;
for(  i = 0; i < total; i++) {
  sum += i;
}
```

Template Literals in ES6

Template literals are a way to output variables in the string.
Prior to ES6 we had to break the string, for example:

CODE PLAYGROUND JS

```
let name = 'David';
let msg = 'Welcome ' + name + '!';
console.log(msg);

Tap to edit |
```

ES6 introduces a new way of outputting variable values in strings. The same code above can be rewritten as:

CODE PLAYGROUND **JS**

```
let name = 'David';
let msg = `Welcome ${name}!`;
console.log(msg);

Tap to edit
```

Notice, that template literals are enclosed by the **backtick** (`` ` ``) character instead of double or single quotes.
The **${expression}** is a placeholder, and can include any expression, which will get evaluated and inserted into the template literal.

For example:

CODE PLAYGROUND **JS**

```
let a = 8;
let b = 34;
let msg = `The sum is ${a+b}`;
console.log(msg);

Tap to edit |
```

To escape a backtick in a template literal, put a backslash \ before the backtick.

Fill in the blanks to output "We are learning ES6!".

```
let n = 6;

let s = 'ES';

let msg = `We are learning ${s + n}!`;

console.log(msg);
```

Loops in ECMAScript 6

In JavaScript we commonly use the **for** loop to iterate over values in a list:

CODE PLAYGROUND **JS**

```
let arr = [1, 2, 3];
for (let k = 0; k < arr.length; k++) {
  console.log(arr[k]);
}

Tap to edit |
```

The **for...in** loop is intended for iterating over the enumerable keys of an object.

For example:

CODE PLAYGROUND **JS**

```
let obj = {a: 1, b: 2, c: 3};
for (let v in obj) {
  console.log(v);
}

Tap to edit
```

The **for...in** loop should **NOT** be used to iterate over arrays because, depending on the JavaScript engine, it could iterate in an arbitrary order. Also, the iterating variable is a **string**, not a number, so if you try to do any math with the variable, you'll be performing string concatenation instead of addition.

Fill in the blanks to iterate through all the characters using the for...of loop.

```
(let ch  "SoloLearn") {
  console.log(ch);
}
```

Functions in ECMAScript 6

Prior to ES6, a JavaScript function was defined like this:

CODE PLAYGROUND JS

```
function add(x, y) {
  var sum = x+y;
  console.log(sum);
}

Tap to edit |
```

ES6 introduces a new syntax for writing functions. The same function from above can be written as:

CODE PLAYGROUND **JS**

```
const add = (x, y) => {
  let sum = x + y;
  console.log(sum);
}

Tap to edit
```

This new syntax is quite handy when you just need a simple function with one argument.
You can skip typing **function** and **return**, as well as some parentheses and braces.

For example:

CODE PLAYGROUND **JS**

```
const greet = x => "Welcome " + x;

Tap to edit
```

The code above defines a function named **greet** that has one argument and returns a message.

If there are no parameters, an empty pair of

If there are no parameters, an empty pair of parentheses should be used, as in

CODE PLAYGROUND JS

```
const x = () => alert("Hi");

Tap to edit
```

The syntax is very useful for inline functions. For example, let's say we have an array, and for each element of the array we need to execute a function. We use the **forEach** method of the array to call a function for each element:

CODE PLAYGROUND JS

```
var arr = [2, 3, 7, 8];

arr.forEach(function(el) {
  console.log(el * 2);
});

Tap to edit
```

However, in ES6, the code above can be rewritten as following:

CODE PLAYGROUND **JS**

```
const arr = [2, 3, 7, 8];

arr.forEach(v => {
  console.log(v * 2);
});

Tap to edit |
```

The code is shorter and looks pretty nice, doesn't it? :)

Fill in the blanks to declare an arrow function that takes an array and prints the odd elements.

```
const printOdds = (arr) ▢ {
  ▢.forEach(▢ => {
    if (el % 2 != 0) console.log(el);
  });
}
```

Default Parameters in ES6

In ES6, we can put the default values right in the signature of the functions.

For example:

CODE PLAYGROUND JS

```
function test(a, b = 3, c = 42) {
  return a + b + c;
}
console.log(test(5)); //50
```

Tap to edit

And here's an example of an arrow function with default parameters:

```JS
const test = (a, b = 3, c = 42) => {
  return a + b + c;
}
console.log(test(5)); //50
```

Default value expressions are evaluated at function call time from left to right. This also means that default expressions can use the values of previously-filled parameters.

What is the output of this code?

```
function magic(a, b = 40) {
  return a + b;
}
console.log(magic(2));
```

ES6 Objects

JavaScript variables can be **Object** data types that contain many values called **properties**.
An object can also have properties that are function definitions called **methods** for performing actions on the object.
ES6 introduces **shorthand** notations and **computed** property names that make declaring and using objects easier to understand.

The new method definition shorthand does not require the colon (:) or **function** keyword, as in the **grow** function of the **tree** object declaration:

CODE PLAYGROUND **JS**

```
let tree = {
  height: 10,
  color: 'green',
  grow() {
    this.height += 2;
  }
};
tree.grow();
console.log(tree.height); // 12

Tap to edit |
```

You can also use a property value shorthand when initializing properties with a variable by the same name.

For example, properties **height** and **health** are being initialized with variables named **height** and **health**:

CODE PLAYGROUND JS

```
let height = 5;
let health = 100;

let athlete = {
  height,
  health
};

Tap to edit |
```

When creating an object by using duplicate property names, the last property will overwrite the prior ones of the same name.

For Example:

CODE PLAYGROUND JS

```
var a = {x: 1, x: 2, x: 3, x: 4};

Tap to edit |
```

Duplicate property names generated

Duplicate property names generated a **SyntaxError** in ES5 when using strict mode. However, ES6 removed this limitation.

Fill in the blanks to make this code run and print 60.

```
let car = {
  speed: 40,
  accelerate() {
    ▢.speed += 10;
  }
};
▢.accelerate();
car.accelerate();
console.log(car.▢);
```

Computed Property Names

With ES6, you can now use **computed property** names. Using the square bracket notation [], we can use an expression for a property name, including concatenating strings. This can be useful in cases where we want to create certain objects based on user data (e.g. id, email, and so on).

Example 1:

CODE PLAYGROUND JS

```
let prop = 'name';
let id = '1234';
let mobile = '08923';

let user = {
  [prop]: 'Jack',
  [`user_${id}`]: `${mobile}`
};

Tap to edit |
```

Example 2:

CODE PLAYGROUND JS

```
var i = 0;
var a = {
  ['foo' + ++i]: i,
  ['foo' + ++i]: i,
  ['foo' + ++i]: i
};
```

It is very useful when you need to create custom objects based on some variables.

Fill in the blanks to create an object with its properties.

```
let prop = 'foo';
let o = {
  [prop]: 'lol',
  ['b' + 'ar']: '123'
};
```

Object.assign() in ES6

ES6 adds a new **Object** method **assign()** that allows us to combine multiple sources into one target to create a single new object. **Object.assign()** is also useful for creating a duplicate of an existing object.

Let's look at the following example to see how to combine objects:

CODE PLAYGROUND **JS**

```
let person = {
  name: 'Jack',
  age: 18,
  sex: 'male'
};
let student = {
  name: 'Bob',
  age: 20,
  xp: '2'
};
let newStudent = Object.assign({},
person, student);

Tap to edit
```

Here we used **Object.assign()** where the first parameter is the **target object** you want to apply new properties to.

Every parameter after the first will be used as **sources** for the target. There are no limitations on the number of source parameters. However, order is important because properties in the second parameter will be overridden by properties of the same name in third parameter, and so on.

Try changing the order of second and third parameters to see what happens to the result.

Now, let's see how we can use **assign()** to create a duplicate object without creating a reference (mutating) to the base object.
In the following example, assignment was used to try to generate a new object. However, using = creates a reference to the base object. Because of this reference, changes intended for a new object mutate the original object:

CODE PLAYGROUND JS

```
let person = {
  name: 'Jack',
  age: 18
};

let newPerson = person; //  newPerson
references person
newPerson.name = 'Bob';

console.log(person.name); // Bob
console.log(newPerson.name); // Bob

Tap to edit
```

To avoid this (mutations), use **Object.assign()** to create a new object.

For example:

CODE PLAYGROUND JS

```
let person = {
  name: 'Jack',
  age: 18
};

let newPerson = Object.assign({},
person);
newPerson.name = 'Bob';

console.log(person.name); // Jack
console.log(newPerson.name); // Bob

Tap to edit |
```

Finally, you can assign a value to an object property in the **Object.assign()** statement.

For example:

CODE PLAYGROUND **JS**

```
let person = {
  name: 'Jack',
  age: 18
};

let newPerson = Object.assign({}, person,
{name: 'Bob'});

Tap to edit |
```

What is the output of this code?

```
const obj1 = {
  a: 0,
  b: 2,
  c: 4
};
const obj2 = Object.assign({c: 5, d: 6}, obj1);
console.log(obj2.c, obj2.d);
```

- 4 6
- 0 5
- 5 6

Array Destructuring in ES6

The **destructuring** assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.
ES6 has added a shorthand syntax for destructuring an array.

The following example demonstrates how to unpack the elements of an array into distinct variables:

CODE PLAYGROUND **JS**

```
let arr = ['1', '2', '3'];
let [one, two, three] = arr;

console.log(one); // 1
console.log(two); // 2
console.log(three); // 3

Tap to edit |
```

We can use also destructure an array returned by a function.

We can use also destructure an array returned by a function.

For example:

CODE PLAYGROUND JS

```
let a = () => {
  return [1, 3, 2];
};

let [one, , two] = a();

Tap to edit
```

Notice that we left the second argument's place empty.

The destructuring syntax also simplifies assignment and swapping values:

CODE PLAYGROUND **JS**

```
let a, b, c = 4, d = 8;
[a, b = 6] = [2]; // a = 2, b = 6

[c, d] = [d, c]; // c = 8, d = 4

Tap to edit |
```

Run the code and see how it works!

What is the output of the following code?

```
let names = ['John', 'Fred', 'Ann'];
let [Ann, Fred, John] = names;
console.log(John);
```

- Ann
- John
- Fred
- Error

Object Destructuring in ES6

Similar to Array destructuring, **Object destructuring** unpacks properties into distinct variables.

For example:

CODE PLAYGROUND **JS**

```
let obj = {h:100, s: true};
let {h, s} = obj;

console.log(h); // 100
console.log(s); // true

Tap to edit |
```

We can assign without declaration, but there are some syntax requirements for that:

CODE PLAYGROUND **JS**

```
let a, b;
({a, b} = {a: 'Hello ', b: 'Jack'});

console.log(a + b); // Hello Jack

Tap to edit
```

The () with a **semicolon (;)** at the end are **mandatory** when destructuring without a declaration. However, you can also do it as follows where the () are not required:

CODE PLAYGROUND **JS**

```
let {a, b} = {a: 'Hello ', b: 'Jack'};
console.log(a + b);

Tap to edit
```

You can also assign the object to new variable names.

For example:

CODE PLAYGROUND JS

```
var o = {h: 42, s: true};
var {h: foo, s: bar} = o;

//console.log(h); // Error
console.log(foo); // 42

Tap to edit |
```

Finally you can assign **default values** to variables, in case the value unpacked from the object is undefined.

For example:

CODE PLAYGROUND **JS**

```
var obj = {id: 42, name: "Jack"};

let {id = 10, age = 20} = obj;

console.log(id); // 42
console.log(age); // 20

Tap to edit |
```

Run the code and see how it works!

What is the output of the following code?

```
const obj = {one: 1, two: 2};
let {one:first, two:second} = obj;
console.log(one);
```

- `1`
- `the full object (obj)`
- `Error`
- `2`

ES6 Rest Parameters

Prior to ES6, if we wanted to pass a variable number of arguments to a function, we could use the **arguments** object, an array-like object, to access the parameters passed to the function. For example, let's write a function that checks if an array contains all the arguments passed:

CODE PLAYGROUND JS

```
function containsAll(arr) {
  for (let k = 1; k < arguments.length;
k++) {
    let num = arguments[k];
    if (arr.indexOf(num) === -1) {
      return false;
    }
  }
  return true;
}
let x = [2, 4, 6, 7];
console.log(containsAll(x, 2, 4, 7));
console.log(containsAll(x, 6, 4, 9));
```

Tap to edit

We can pass any number of arguments to the function and access it using the **arguments** object.

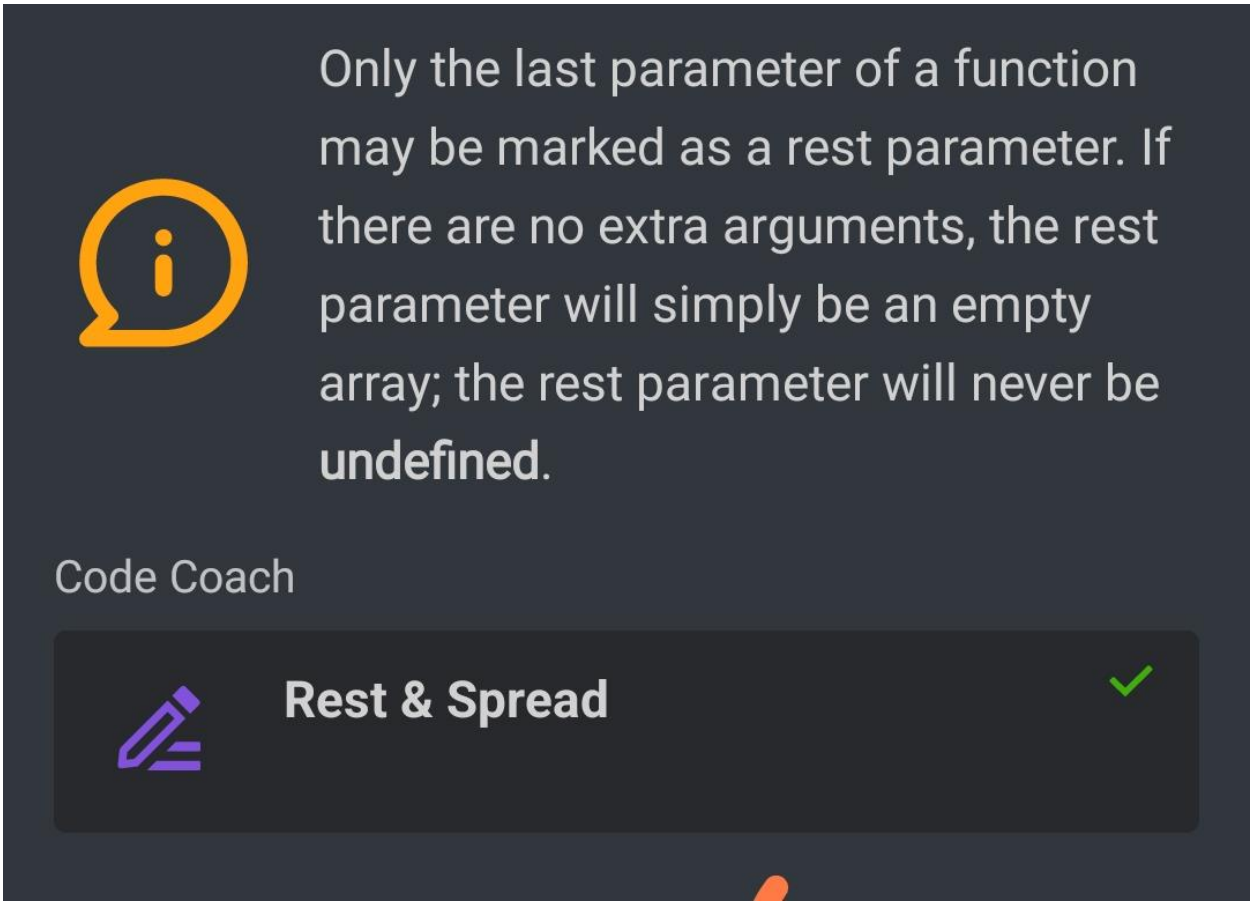
While this does the job, ES6 provides a more readable syntax to achieve variable number of parameters by using a **rest parameter**:

CODE PLAYGROUND **JS**

```
function containsAll(arr, ...nums) {
  for (let num of nums) {
    if (arr.indexOf(num) === -1) {
      return false;
    }
  }
  return true;
}

Tap to edit |
```

The **...nums** parameter is called a **rest parameter**. It takes all the "extra" arguments passed to the function. The three dots (...) are called the **Spread operator**.


Only the last parameter of a function may be marked as a rest parameter. If there are no extra arguments, the rest parameter will simply be an empty array; the rest parameter will never be **undefined**.
Code Coach
Rest & Spread

What is the output of the following code?

```
function magic(...nums) {
  let sum = 0;
  nums.filter(n => n % 2 == 0).map(el => sum+= el);
  return sum;
}
console.log(magic(1, 2, 3, 4, 5, 6));
```

The Spread Operator

This operator is similar to the Rest Parameter, but it has another purpose when used in objects or arrays or function calls (arguments).

[h2]Spread in function calls[/h2]

It is common to pass the elements of an array as arguments to a function. Before ES6, we used the following method:

CODE PLAYGROUND **JS**

```
function myFunction(w, x, y, z) {
  console.log(w + x + y + z);
}
var args = [1, 2, 3];
myFunction.apply(null, args.concat(4));

Tap to edit |
```

ES6 provides an easy way to do the example above with **spread operators**:

CODE PLAYGROUND JS

```
const myFunction = (w, x, y, z) => {
  console.log(w + x + y + z);
};
let args = [1, 2, 3];
myFunction(...args, 4);

Tap to edit |
```

Example:

CODE PLAYGROUND JS

```
var dateFields = [1970, 0, 1];  // 1 Jan
1970
var date = new Date(...dateFields);
console.log(date);

Tap to edit |
```