What is ECMAScript 6 (or ES6)

It is a major enhancement to the JavaScript language, and adds many more features intended to make large-scale software development easier.

Major web browsers support some features of ES6. However, it is possible to use software known as a transpiler (babel) to convert ES6 code into ES5, which is better supported on most browsers.

The let Keyword

var is function-scoped and hoisted at the top within its scope, whereas variables declared with let keyword are block-scoped (`{}`) and they are not hoisted.

Example

Try this code »

```
// ES6 syntax
for(let i = 0; i < 5; i++) {
    console.log(i); // 0,1,2,3,4
}
console.log(i); // undefined


// ES5 syntax
for(var i = 0; i < 5; i++) {
    console.log(i); // 0,1,2,3,4
}
console.log(i); // 5
```

The const Keyword

The new const keyword makes it possible to define constants. Constants are read-only, you cannot reassign new values to them. They are also block-scoped like let.

Example

Try this code »

```
const PI = 3.14;
console.log(PI); // 3.14

PI = 10; // error
```

However, you can still change object properties or array elements:

Example

Try this code »

```
// Changing object property value
const PERSON = {name: "Peter", age: 28};
console.log(PERSON.age); // 28
PERSON.age = 30;
console.log(PERSON.age); // 30

// Changing array element
const COLORS = ["red", "green", "blue"];
console.log(COLORS[0]); // red
COLORS[0] = "yellow";
console.log(COLORS[0]); // yellow
```

The for...of Loop

The new for...of loop allows us to iterate over arrays or other iterable objects very easily. Also, the code inside the loop is executed for each element of the iterable object. Here's an example:

Example

Try this code »

```
// Iterating over array
let letters = ["a", "b", "c", "d", "e", "f"];

for(let letter of letters) {
    console.log(letter); // a,b,c,d,e,f
}

// Iterating over string
let greet = "Hello World!";

for(let character of greet) {
    console.log(character); // H,e,l,l,o, ,W,o,r,l,d,!
}
```

The for...of loop doesn't work with objects because they are not iterable. If you want to iterate over the properties of an object you can use the for...in loop.

Template Literals

Template literals are created using back-tick (`` ` ``) (grave accent) character instead of the usual double or single quotes. Variables or expressions can be placed inside the string using the `${...}` syntax. Compare the following examples and see how much useful it is:

Example

Try this code »

```
// Simple multi-line string
let str = `The quick brown fox
    jumps over the lazy dog.`;

// String with embedded variables and expression
let a = 10;
let b = 20;
let result = `The sum of ${a} and ${b} is ${a+b}.`;
console.log(result); // The sum of 10 and 20 is 30.
```

While in ES5, to achieve the same we had to write something like this:

Example

Try this code »

```
// Multi-line string
var str = 'The quick brown fox\n\t'
    + 'jumps over the lazy dog.';

// Creating string using variables and expression
var a = 10;
var b = 20;
var result = 'The sum of ' + a + ' and ' + b + ' is ' +
(a+b) + '.';
console.log(result); // The sum of 10 and 20 is 30.
```

Default Values for Function Parameters

Now, in ES6 you can specify default values to the function parameters.

Example

Try this code »

```
function sayHello(name='World') {
    return `Hello ${name}!`;
}

console.log(sayHello()); // Hello World!
console.log(sayHello('John')); // Hello John!
```

While in ES5, to achieve the same we had to write something like this:

Example

Try this code »

```
function sayHello(name) {
    var name = name || 'World';
    return 'Hello ' +  name + '!';
}

console.log(sayHello()); // Hello World!
console.log(sayHello('John')); // Hello John!
```

Arrow Functions

Arrow Functions are another interesting feature in ES6. It provides a more concise syntax for writing function expressions by opting out the `function` and `return` keywords.

Arrow functions are defined using a new syntax, the fat arrow (`=>`) notation. Let's see how it looks:

Example
Try this code »

```
// Function Expression
var sum = function(a, b) {
    return a + b;
}
console.log(sum(2, 3)); // 5

// Arrow function
var sum = (a, b) => a + b;
console.log(sum(2, 3)); // 5
```

If there's more than one expression in the function body

Example
Try this code »

```
// Single parameter, single statement
var greet = name => alert("Hi " + name + "!");
greet("Peter"); // Hi Peter!

// Multiple arguments, single statement
var multiply = (x, y) => x * y;
alert(multiply(2, 3)); // 6


// Single parameter, multiple statements
var test = age => {
    if(age > 18) {
        alert("Adult");
    } else {
        alert("Teenager");
    }
}
```

```
test(21); // Adult

// Multiple parameters, multiple statements
var divide = (x, y) => {
    if(y != 0) {
        return x / y;
    }
}
alert(divide(10, 2)); // 5

// No parameter, single statement
var hello = () => alert('Hello World!');
hello(); // Hello World!
```

This

There is an important difference between regular functions and arrow functions. Unlike a normal function, an arrow function does not have its own `this`, it takes `this` from the outer function where it is defined. In JavaScript, `this` is the current execution context of a function.

To understand this clearly, let's check out the following examples:

Example

Try this code »

```
function Person(nickname, country) {
    this.nickname = nickname;
    this.country = country;

    this.getInfo = function() {
        // Outer function context (Person object)
        return function() {
            // Inner function context (Global object
'Window')
            alert(this.constructor.name); // Window
            alert("Hi, I'm " + this.nickname + " from " +
this.country);
        };
    }
}

var p = new Person('Rick', 'Argentina');
var printInfo = p.getInfo();
printInfo(); // Hi, I'm undefined from undefined
```

Rewriting the same example using ES6 template literals and arrow function:

Example

Try this code »

```
function Person(nickname, country) {
    this.nickname = nickname;
    this.country = country;

    this.getInfo = function() {
        // Outer function context (Person object)
        return () => {
            // Inner function context (Person object)
            alert(this.constructor.name); // Person
            alert(`Hi, I'm ${this.nickname} from
${this.country}`);
        };
    }
}

let p = new Person('Rick', 'Argentina');
let printInfo = p.getInfo();
printInfo(); // Hi, I'm Rick from Argentina
```

Classes

In ES6 you can declare a class using the new `class` keyword followed by a class-name. By convention class names are written in TitleCase (i.e. capitalizing the first letter of each word).

Example

Try this code »

```
class Rectangle {
    // Class constructor
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    // Class method
    getArea() {
        return this.length * this.width;
    }
}

// Square class inherits from the Rectangle class
class Square extends Rectangle {
    // Child class constructor
    constructor(length) {
        // Call parent's constructor
        super(length, length);
    }

    // Child class method
    getPerimeter() {
        return 2 * (this.length + this.width);
    }
}

let rectangle = new Rectangle(5, 10);
alert(rectangle.getArea()); // 50

let square = new Square(5);
alert(square.getArea()); // 25
alert(square.getPerimeter()); // 20

alert(square instanceof Square); // true
alert(square instanceof Rectangle); // true
alert(rectangle instanceof Square); // false
```

In the above example the Square class inherits from Rectangle using the `extends` keyword. Classes that inherit from other classes are referred to as derived classes or child classes.

Also, you must call `super()` in the child class constructor before accessing the context (`this`). For instance, if you omit the `super()` and call the `getArea()` method on square object it will result in an error, since `getArea()` method require access to `this` keyword.

Note: Class declarations are not hoisted.

Modules

ES6 introduces file based module, in which each module is represented by a separate `.js` file. Now, you can use the `export` or `import` statement in a module to export or import variables, functions, classes or any other entity to/from other modules or files.

Let's create a module i.e. a JavaScript file "main.js" and place the following code in it:

Example
Try this code »

```
let greet = "Hello World!";
const PI = 3.14;

function multiplyNumbers(a, b) {
    return a * b;
}

// Exporting variables and functions
export { greet, PI, multiplyNumbers };
```

Now create another JavaScript file "app.js" with the following code:

Example
Try this code »

```
import { greet, PI, multiplyNumbers } from './main.js';

alert(greet); // Hello World!
alert(PI); // 3.14
alert(multiplyNumbers(6, 15)); // 90
```

Finally create a HTML file "test.html" and with the following code and open this HTML file in your browser using HTTP protocol (or use localhost). Also notice the `type="module"` on script tag.

Example
Try this code »

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="utf-8">
    <title>ES6 Module Demo</title>
</head>
<body>
    <script type="module" src="app.js"></script>
</body>
</html>
```

The Rest Parameters

A rest parameter is specified by prefixing a named parameter with rest operator (`...`) i.e. three dots. Rest parameter can only be the last one in the list of parameters, and there can only be one rest parameter.

Example

Try this code »

```
function sortNames(...names) {
    return names.sort();
}

alert(sortNames("Sarah", "Harry", "Peter")); //
Harry,Peter,Sarah
alert(sortNames("Tony", "Ben", "Rick", "Jos")); //
John,Jos,Rick,Tony
```

Example

Try this code »

```
function myFunction(a, b, ...args) {
    return args;
}

alert(myFunction(1, 2, 3, 4, 5)); // 3,4,5
alert(myFunction(-7, 5, 0, -2, 4.5, 1, 3)); // 0,-2,4.5,1,3
```

The Spread Operator

The spread operator, which is also denoted by (`...`), performs the exact opposite function of the rest operator.

Example

Try this code »

```
function addNumbers(a, b, c) {
    return a + b + c;
}

let numbers = [5, 12, 8];

// ES5 way of passing array as an argument of a function
alert(addNumbers.apply(null, numbers)); // 25

// ES6 spread operator
alert(addNumbers(...numbers)); // 25
```

The spread operator can also be used to insert the elements of an array into another array without using the array methods like `push()`, `unshift()` `concat()`, etc.

Example

Try this code »

```
let pets = ["Cat", "Dog", "Parrot"];
let bugs = ["Ant", "Bee"];

// Creating an array by inserting elements from other
arrays
let animals = [...pets, "Tiger", "Wolf", "Zebra", ...bugs];

alert(animals); // Cat,Dog,Parrot,Tiger,Wolf,Zebra,Ant,Bee
```

Destructuring Assignment

The destructuring assignment is an expression that makes it easy to extract values from arrays, or properties from objects, into distinct variables by providing a shorter syntax.

There are two kinds of destructuring assignment expressions—
the *array* and *object* destructuring assignment.

The array destructuring assignment

Prior to ES6, to get an individual value of an array we need to write something like this:

Example
Try this code »

```
// ES5 syntax
var fruits = ["Apple", "Banana"];

var a = fruits[0];
var b = fruits[1];
alert(a); // Apple
alert(b); // Banana
```

In ES6, we can do the same thing in just one line using the array destructuring assignment:

Example
Try this code »

```
// ES6 syntax
let fruits = ["Apple", "Banana"];

let [a, b] = fruits; // Array destructuring assignment

alert(a); // Apple
alert(b); // Banana
```

You can also use rest operator in the array destructuring assignment, as shown here:

Example

Try this code »

```
// ES6 syntax
let fruits = ["Apple", "Banana", "Mango"];

let [a, ...r] = fruits;

alert(a); // Apple
alert(r); // Banana,Mango
alert(Array.isArray(r)); // true
```

The object destructuring assignment

In ES5 to extract the property values of an object we need to write something like this:

Example

Try this code »

```
// ES5 syntax
var person = {name: "Peter", age: 28};

var name = person.name;
var age = person.age;

alert(name); // Peter
alert(age); // 28
```

But in ES6, you can extract object's property values and assign them to the variables easily like this:

Example

Try this code »

```
// ES6 syntax
let person = {name: "Peter", age: 28};

let {name, age} = person; // Object destructuring
assignment

alert(name); // Peter
alert(age); // 28
```

Most of the features we've discussed above are supported in the latest version of the major web browsers such as Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, etc.

Alternatively, you can use the online transpilers (source-to-source compilers) like Babel free of cost to transpile your current ES6 code to ES5 for better browser compatibility without leaving out the benefits of enhanced syntax and capabilities of ES6.