# ECMAScript 6 also known as ES6

## JavaScript let

The `let` statement allows you to declare a variable with block scope.

```
var x = 10;
// Here x is 10
{
  let x = 2;
  // Here x is 2
}
// Here x is 10
```

## JavaScript const

The `const` statement allows you to declare a constant (a JavaScript variable with a constant value).

Constants are similar to let variables, except that the value cannot be changed.

```
var x = 10;
// Here x is 10
{
  const x = 2;
  // Here x is 2
}
// Here x is 10
```

## Exponentiation Operator

The **exponentiation** operator (`**`) raises the first operand to the power of the second operand.

```
var x = 5;
var z = x ** 2;          // result is 25
```

`x ** y` produces the same result as `Math.pow(x,y)`:

## Default Parameter Values

ES6 allows function parameters to have default values.

```
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
myFunction(5); // will return 15
```

## New Number Properties

ES6 added the following properties to the Number object:

- `EPSILON`
- `MIN_SAFE_INTEGER`
- `MAX_SAFE_INTEGER`

```
var x = Number.EPSILON;

// value of x : 2.220446049250313e-16

var x = Number.MIN_SAFE_INTEGER;

// value of x : -9007199254740991

var x = Number.MAX_SAFE_INTEGER;

// value of x : 9007199254740991
```

## New Number Methods

ES6 added 2 new methods to the Number object:

- `Number.isInteger()`
- `Number.isSafeInteger()`

## The Number.isInteger() Method

The `Number.isInteger()` method returns `true` if the argument is an integer.

### Example

```
Number.isInteger(10);        // returns true
Number.isInteger(10.5);      // returns false
```

## The Number.isSafeInteger() Method

A safe integer is an integer that can be exactly represented as a double precision number.

The `Number.isSafeInteger()` method returns `true` if the argument is a safe integer.

### Example

```
Number.isSafeInteger(10);    // returns true
Number.isSafeInteger(12345678901234567890);  // returns false
```

Safe integers are all integers from -($2^{53}$ - 1) to +($2^{53}$ - 1).
This is safe: 9007199254740991. This is not safe: 9007199254740992.

## New Global Methods

ES6 also added 2 new global number methods:

- `isFinite()`
- `isNaN()`

---

## The isFinite() Method

The global `isFinite()` method returns `false` if the argument is `Infinity` or `NaN`.

Otherwise it returns `true`:

### Example

```
isFinite(10/0);       // returns false
isFinite(10/1);       // returns true
```

## The isNaN() Method

The global `isNaN()` method returns `true` if the argument is `NaN`. Otherwise it returns `false`:

### Example

```
isNaN("Hello");       // returns true
```

## Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the `function` keyword, the `return` keyword, and the **curly brackets**.

### Example

```
// ES5
var x = function(x, y) {
   return x * y;
}

// ES6
const x = (x, y) => x * y;
```

Arrow functions do not have their own `this`. They are not well suited for defining **object methods**.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using `const` is safer than using `var`, because a function expression is always constant value.

You can only omit the `return` keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

Example

```
const x = (x, y) => { return x * y };
```

## Template literals

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

```
`string text`


`string text line 1

 string text line 2`


`string text ${expression} string text`


tag `string text ${expression} string text`
```

## Destructuring array and objects

**Destructuring** in JavaScript is a simplified method of extracting multiple properties from an array by taking the structure and deconstructing it down into its own constituent parts through assignments by using a syntax that looks similar to array literals.

```
let numbers = [1,2,3];

let [a,b] = numbers;



console.log(a); // 1

console.log(b); // 2
```

```
const person = {

  first: 'Wes',

  last: 'Bos',
  country: 'Canada',

  city: 'Hamilton',

  twitter: '@wesbos'

};

const { first, last } = person;


console.log(first); // Wes

console.log(last); // Bos
```

## Rest and Spread

Rest parameters are indicated by three dots **…** preceding a parameter. Named parameter becomes an **array** which contain the rest of the parameters.

```
function sumUp(...toAdd) {

            return toAdd;

}

console.log(sumUp(2,3));

 // [2,3]
```

The **spread** is closely related to rest parameters, because of **…** (three dots) notation. It allows to split an array to single arguments which are passed to the function as separate arguments.

```
let numbers = [1,2,3];

console.log(Math.max(...numbers));

// 3
```