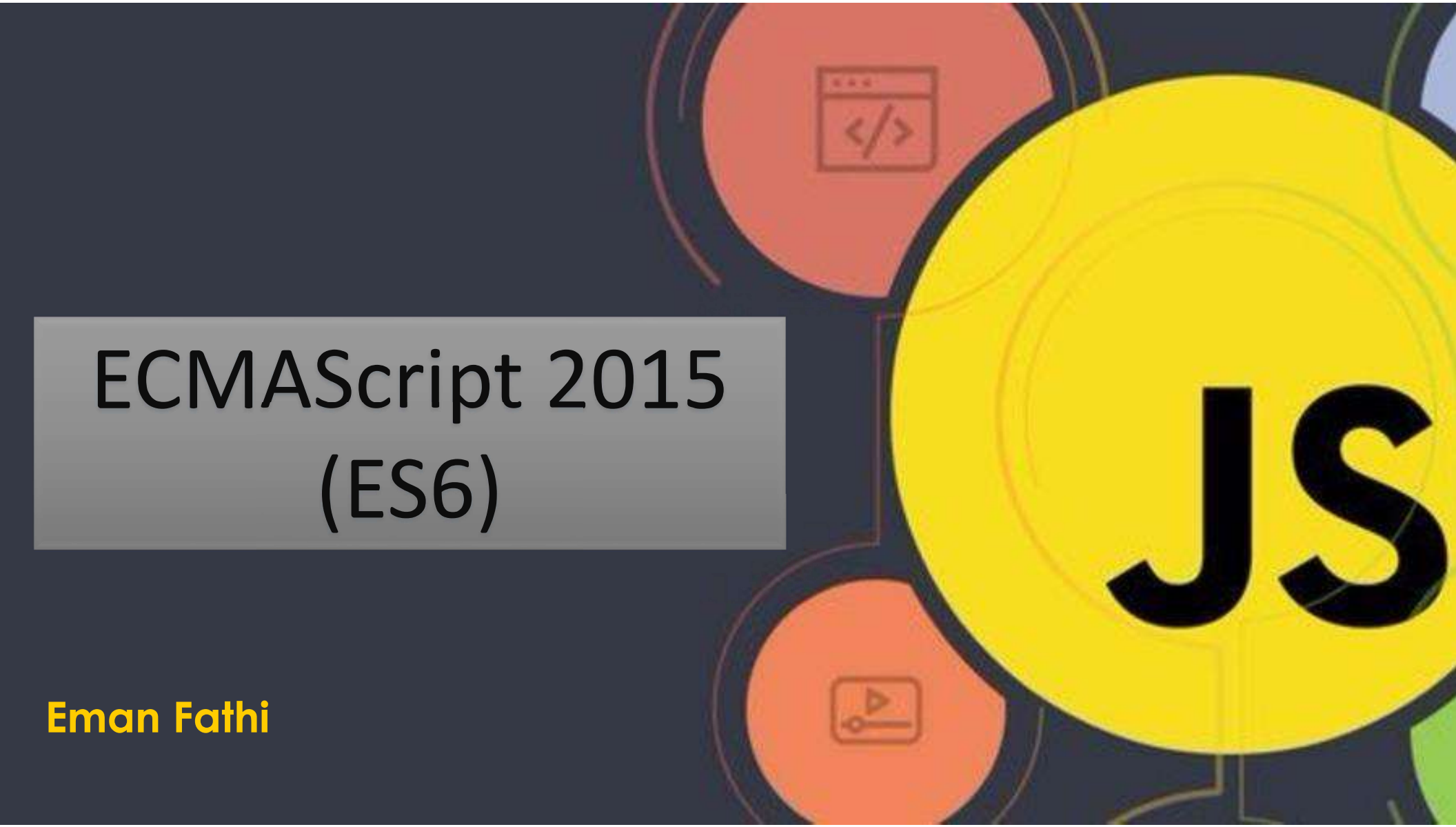
ECMAScript 2015 (ES6)



Eman Fathi

ECMAScript Releases

JavaScript is born as LiveScript

1997

ES3 comes out and IE5 is all the rage

2000

ES5 comes out and standard JSON

2015

ES7/ECMAScript2016 comes out

2017

1995 **ECMAScript** standard is established

1999 XMLHttpRequest, a.k.a. AJAX, gains popularity

2009 **ES6**/ECMAScript2015 comes out

2016 ES.Next


ADVANTAGES OF JAVASCRIPT
ES6
MODULES
ARROWS
ENHANCED OBJECT LITERALS
CLASSES
BLOCK SCOPING
PROMISES

ECMAScript 5 6 2016+ next intl non-standard compatibility table

Flattr | by kangax & webbedspace & zloirock | Fork 574

Sort by: Engine types | Show obsolete platforms | Show unstable platforms

V8 | SpiderMonkey | JavaScriptCore | Chakra | Carakan | KJS | Other

Minor difference (1 point) | Small feature (2 points) | Medium feature (4 points) | Large feature (8 points)

		Compilers/polyfills						Desktop browsers												
	97%	56%	71%	71%	49%	59%	17%	5%	11%	96%	96%	96%	94%	97%	97%	97%	98%	97%	97%	98%
Feature name	Current browser	Traceur	Babel 6 + core-js[2]	Babel 7 + core-js[2]	Closure	TypeScript + core-js	es6-shim	Konq 4.14[3]	IE 11	Edge 15	Edge 16	Edge 17 Preview	FF 52 ESR	FF 57	FF 58	FF 59 Beta	FF 60 Nightly	CH 64, OP 51[1]	CH 65, OP 52[1]	CH 66, OP 53[1]
Optimisation																				
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
Syntax																				
default function parameters	7/7	4/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	6/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7
rest parameters	5/5	4/5	3/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
spread (...) operator	15/15	15/15	13/15	13/15	12/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15
object literal extensions	6/6	6/6	6/6	6/6	5/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
for..of loops	9/9	9/9	9/9	9/9	6/9	3/9	0/9	0/9	0/9	9/9	9/9	9/9	7/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9
octal and binary literals	4/4	2/4	4/4	4/4	4/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
template literals	5/5	4/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
RegExp "y" and "u" flags	5/5	3/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
destructuring, declarations	22/22	20/22	21/22	21/22	20/22	15/22	0/22	0/22	0/22	22/22	22/22	22/22	21/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22
destructuring, assignment	24/24	23/24	24/24	24/24	21/24	19/24	0/24	0/24	0/24	24/24	24/24	24/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
destructuring, parameters	24/24	19/24	21/24	21/24	19/24	16/24	0/24	0/24	0/24	23/24	23/24	23/24	21/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
new.target	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Bindings																				
const	16/16	14/16	14/16	14/16	14/16	14/16	0/16	2/16	12/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16
let	12/12	10/12	10/12	10/12	10/12	10/12	0/12	0/12	10/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12

ECMAScript 2015 (ES6)

- Since ECMAScript 2015 (also known as ES6) was released, it has introduced a huge set of new features. They include arrow functions, sets, maps, classes and destructuring, and much more. In many ways, ES2015 is almost like learning a new version of JavaScript.
- Ecma Technical Committee governs the ECMA specification. They decided to release a new version of ECMAScript every year starting in 2015. A yearly update means no more big releases like ES6.

Top 10 best ES6 Features

- ✓Block scope constructs with let and const
- ✓Template Literals
- ✓Multi-Line strings
- ✓Arrow functions
- ✓Default parameters
- ✓Enhanced object literals
- ✓Classes
- ✓Destructing Assignment
- ✓Modules
- ✓Promises

Let and Const Block Scoping

Block Level Declaration with let and const

Variable declarations using **var** are treated as if they're at the top of the function (or in the global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called **hoisting**.

Misunderstanding hoisting unique behaviour can end up causing bugs. For this reason, ECMAScript 6 introduces **block-level** scoping options to give developers more control over a variable's life cycle.

Block Scopes are created in the following places:

1-Inside a function

2- Inside a block (indicated by the { and } characters)

Let Declaration

The **let** declaration syntax is the same as the syntax for **var**. You can basically replace var with let to declare a variable but limit the variable's scope to only the current code block

```
let studentId = 2;
console.log(studentId); // output  2
```

let declarations are **not hoisted** to the top of the enclosing block, it's best to place let declarations first in the block so they're available to the entire block.

```
console.log(studentId); //error: studentId is not defined
let studentId = 2;
```

By using let you **can not** define parameter twice

```
let count = 2;
var count=2; //error : Identifier 'count' has already been declared
//or
var count=2;
let count = 2; //error : Identifier 'count' has already been declared
```

Because let will not redefine an identifier that already exists in the same scope, the let declaration will throw an error.

```
var count = 30;
if (true) {
            let count = 40; // doesn't throw an error
          }
function countWords(myName)
{
            let count = 0;
            //more code
}
```

Const Declaration

constants, meaning their values **cannot be** changed once set. For this reason, every const variable must be initialized on declaration

```
// valid constant
const maxItems = 30;
maxItems=40; //error: Assignment to constant variable.
const name; // syntax error: Missing initializer in const declaration
```

Const variables are not hoisted

```
console.log(maxItems); //error : maxItems is not defined
const maxItems = 30;
```

✓Constants, like let declarations, are block-level declarations.

```
const maxItems = 30;
if (true) {
        const maxItems = 5;
        // more code
    }
```

✓In another similarity to let, a const declaration throws an error when made with an identifier for an already defined variable in the same scope.

```
var message = "Hello!";
let age = 25;
// each of these throws an error
const message = "Goodbye!";
const age = 30;
```

✓Even if we start defining variable with const

```
const age = 30;
let age=30;// error : Identifier 'age' has already been declared
```

Block Bindings in Loops

Perhaps one area where developers most want block-level scoping of variables is within , where the throwaway counter variable is meant to be used only inside the loop.

items=[1,2,3,4,5,6,7,8,9,10]

```
for (var i = 0; i < 10; i++) {
    console.log (items[i]);
}
console.log(i); // 10 -> i is still accessible here
```

The variable i is still accessible after the loop is completed because the var declaration is hoisted.

```
for (let i = 0; i < 10; i++) {
    console.log (items[i]);
}
 console.log(i); // i is not accessible here - throws an error
```

In this example, the variable i exists only within the for loop. When the loop is complete, the variable is no longer accessible elsewhere.

Functions in loops

```
<ul>
    <li>water</li>
    <li>milk</li>
</ul>
<script>
    var liElm = document.getElementsByTagName("li");
    for (var i = 0;i < liElm.length; i++) {
        liElm[i].onclick = function () {
            alert(i); //alert always show 2
        };
    }
</script>
```

The characteristics of var have long made creating functions inside loops problematic, because the loop variables are accessible from outside the scope of the loop.

The let declaration creates a **new** variable **i** each time through the loop, so each function created inside the loop gets its own copy of i. Each copy of i has the value it was assigned at the beginning of the loop iteration in which it was created.

```
for (let i = 0;i < liElm.length; i++) {
        liElm[i].onclick = function () {
            alert(i);
        };
    }
```

The same is true for for-in and for-of loops.

Global Blocks Bindings

Another way in which let and const are different from var is in their **global scope** behaviour. When var is used in the global scope, it creates a new global variable, which is a property on the global object (window in browsers). That means you can accidentally overwrite an existing global using var

```
var name="Eman";

console.log(name===window.name); // true
```

If you instead use let or const in the global scope, a new binding is created in the global scope but no property is added to the global object.

```
let name="Eman";

console.log(name===window.name); // false
```

Template Literals & multiline String

Template Literals

Template literals are ECMAScript 6's answer to the following features that JavaScript lacked in ECMAScript 5 and in earlier versions:

✓Multiline strings A formal concept of multiline strings

✓Basic string formatting The ability to substitute parts of the string for values contained in variables.

At their simplest, template literals act like regular strings delimited by **backticks (`)** instead of double or single quotes.

```
let message = `Hello world!`;
console.log(message); // "Hello world!"
console.log(typeof message); // "string"
console.log(message.length); // 12
```

There's no need to escape either double or single quotes inside template literals.

Multiline Strings

JavaScript developers have wanted a way to create multiline strings since the first version of the language. But when you're using double or single quotes, strings must be completely contained on a single line.

Pre-ECMAScript 6 Workarounds Thanks to a long-standing syntax bug, JavaScript does have a workaround for creating multiline strings. You can create multiline strings by using a **backslash (\)** before a newline.

```
var message = "Multiline \
string";
console.log(message); // "Multiline string“
```

ECMAScript 6's template literals make multiline strings easy because there's no special syntax. Just include a newline where you want, and it appears in the result

```
let message = `Multiline
string`;
console.log(message); // "Multiline
                         // string"
console.log(message.length); // 16
```

All whitespace inside the backticks is part of the string, so be careful with indentation. For example:

```
let message = `Multiline
               string`;
console.log(message); // "Multiline
                                    // string"
console.log(message.length); // 31
```

Making Substitutions

Substitutions are delimited by an opening ${ and a closing } that can have any JavaScript expression inside. The simplest substitutions let you embed local variables directly into a resulting string.

```
let name = "Nicholas",
message = `Hello, ${name}.`;
console.log(message); // "Hello, Nicholas.“
```

Because all substitutions are JavaScript expressions, you can substitute more than just simple variable names. You can easily embed calculations, function calls, and more.

```
let count = 10,
price = 0.25,
message = `${count} items cost $${(count * price).toFixed(2)}.`;
console.log(message); // "10 items cost $2.50."
```

Functions Blocks

Functions Blocks

- ✓ Functions with Default Parameter Values
- ✓ Arrow Functions
- ✓ Spread and rest Operators

Function Blocks: Default Parameters

Functions with Default Parameter Values

Functions in JavaScript are **unique** in that they allow any number of parameters to be passed regardless of the number of parameters declared in the function definition. This allows you to define functions that can handle different numbers of parameters, often by just filling in **default values** when parameters aren't provided.

```
function getProduct(price, type="HardWare")
{
    //do something
}
//uses default type parameter
getProduct(1000);
//overwrite Default type value
getProduct(1000,"software");
```

```
function makeRequest(url, timeout = 2000, callback = function() {}) {
    // the rest of the function };

// uses default timeout and callback
makeRequest("/foo");
// uses default callback
makeRequest("/foo", 500);
// doesn't use defaults
makeRequest("/foo", 500, function() {
//doSomething; });
function callBackRequest(){/*body*/}
makeRquest("/foo",500,callBackRequest);
```

How Default Parameter Values Affect the arguments Object

```
function makeRequest(url, timeout = 2000, callback = function() {})
                                                {
                                                    console.log(arguments.length); };


// uses default timeout and callback
makeRequest("/foo"); //→ output 1
// uses default callback
makeRequest("/foo", 500); //→ output 2
// doesn't use defaults
makeRequest("/foo", 500, function() {
//doSomething; });       //→ output 3
```

Default Parameter Expressions

The most interesting feature of default parameter values is that the default value need not be a primitive value.

```
let baseDiscount=0.5;
function getProduct(price, type="HardWare",Discount=baseDiscount)
{   //do somthing ; }
getProduct(1000); //Discount→ 0.5
function getBaseDiscount(){ return 0.2 }
function getProduct(price, type="HardWare",Discount=getBaseDiscount())
{  //do something  }
getProduct(1000); //Discount→ 0.2
```

Keep in mind that **getBaseDiscount()** is called only when **getProduct()** is called without a second parameter

You can use a previous parameter as the default for a later parameter.

```
function getProduct(price, type="HardWare",Discount=price*0.2)
{   //do something   }
getProduct(1000); // Discount = 200
```

you can pass **price** into a function to get the value for **Discount**

```
function getBaseDiscount(value){ return value*0.2 }
function getProduct(price, type="HardWare",Discount=getBaseDiscount(price))
{  //do something  }
getProduct(1000); //Discount = 200
```

The ability to reference parameters from default parameter assignments works only for **previous** arguments, so earlier arguments don't have access to later arguments.

```
function getProduct(price=Discount, type="HardWare",Discount=0.3)
{  //do something  }
getProduct(); // ERROR
```

Function Blocks: Rest and Spread

Rest Parameters

A **rest** parameter is indicated by **three dots (...)** preceding a named parameter. That named parameter becomes an **Array** containing the rest of the parameters passed to the function, which is where the name rest parameters originates.

```
function showProducts(orderId,...products)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
showProducts(2,"item1","item2");
```

Rest Parameter Restrictions

The first restriction is that there **can be only one rest parameter**, and the rest parameter **must be last parameter**.

```
function showProducts(orderId,...products,category)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
```

Or

```
function showProducts(orderId,...products,...category)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
```

Syntax Error → Rest parameter must be last formal parameter.

The Spread Operator

The **spread** operator allows you to specify an **array** that should be **split** and passed in as separate arguments to a function.

Consider the built-in Math.max() method, which accepts any number of arguments and returns the one with the highest value.

```
var numbers = [3, 1, 7, 4, 9];
console.log(Math.max(3,1,7,4,9)); //9
//or
console.log(Math.max.apply(null,numbers)); //9
```

you can pass the array to Math.max() directly and prefix it with the same **... pattern** you use with rest parameters. The JavaScript engine then splits the array into individual arguments and passes them in.

```
console.log(Math.max(…numbers)); //9
```

We can use spread inside another array as follow

```
var AllNumbers = [33, 55, 11, ...numbers, 90]; // [33,55,11,3,1,7,4,9,90]
```

Function Blocks: Arrow Functions

Arrow Functions

One of the **most** interesting new parts of **ECMAScript 6** is the arrow function.

Arrow functions are, as the name suggests, functions defined with a new syntax that uses an **arrow (=>)**. But arrow functions behave differently than traditional JavaScript functions in a number of important ways:

- **Cannot be called with new** Arrow functions do not have a [[Construct]] method and therefore cannot be used as constructors. Arrow functions throw an error when used with new.
- **Can't change this** The value of this inside the function can't be changed. It remains the same throughout the entire life cycle of the function.
- **No arguments object** Because arrow functions have no arguments binding, you must rely on named and rest parameters to access function arguments.

Arrow Function Syntax

All variations begin with function arguments, followed by the arrow, followed by the body of the function. The arguments and the body can take different forms depending on usage.

Function have no input or return

```
let getPrice = () => console.log("testing");
getPrice(); //-->testing
```

This is equivalent to

```
let getPrice =function ()
{
    console.log("testing");
}
```

✓Function take one argument and return one value

```
let getBaseDiscount = (price) => price*0.2;
console.log(getBaseDiscount (1000)); //-->200
```

This is equivalent to

```
let getBaseDiscount=function (price) {
    return price * 0.2
}
```

✓Function with more than one input and return one value

```
let getPrice = (product, price) =>[product, price];
```

✓Function with more than one output statement

```
let getPrice = (product, price) => {
            let result;
            if(price>50)
                result = product + " : " + (price * 2)
            else
                result = product + " : " + (price)
            return result;
        }
```

Curly braces denote the function's body, which works just fine in the cases you've seen so far. But an arrow function that wants to return an object literal outside a function body must wrap the literal in parentheses.

```
let getTempItem = id => ({ id: id, name: "Temp" });
// effectively equivalent to:
let getTempItem = function(id) {
    return {
        id: id,
        name: "Temp"
    };
};
```

No this Binding

Because the value of this can change inside a single function depending on the context in which the function is called, it's possible to mistakenly affect one object when you meant to affect another.

```
function Person()
{    this.count=20;
    setTimeout(function(){
    this.count++;  //this here does not refer to Person Object (Window Object)!!!
    },2000);
}
This is equivalent to
function timeoutFun()
{ this.count++; //here this refers to window object }
function Person()
{    this.count=20;     setTimeout("timeoutFun()",2000);}
```

Arrow functions have no this binding, which means the value of this inside an arrow function can only be determined by looking up the scope chain.

```
let pageHandler={
    id:1234,
    init:function(){
        window.addEventListener("click",function(){
            //some intialization code
            this.afterLoading();  //this here refers to documentObject
        });
    },
    afterLoading:function(){
        console.log("afterLoading");
    }
};
pageHandler.init() //error afterLoading is not a function
```

Now using arrow function

```
pageHandler={
    id:1234,
    init:function(){
        window.addEventListener("click",()=>{
            //some intialization code
            this.afterLoading();
        });
    },
    afterLoading:function(){
        console.log("afterLoading");
    }
};
```

Warnings

```
var Book = {Title: "ES",
            Borrow: function (name) {console.log(name+ "borrowing"+ this.Title);}
            };
console.log(Book.Borrow("Eman"));    // Eman borrowing ES
```

But

```
var Book = {Title: "ES",
            Borrow:(name) =>{ console.log(name + " borrowing " + this.Title); }
            };
Console.log(Book.Borrow("Eman"));    // Eman borrowing undefined
```

And

If you try to use the new operator with an arrow function, you'll get an error

```
var MyType = () => {},
object = new MyType(); // error - MyType is not a constructor
```

Arrow Functions and Arrays

The concise syntax for arrow functions makes them ideal for use with array processing, too.

```
var result = values.sort(function(a, b) {
    return a - b;
});


var result = values.sort((a, b) => a - b);
```

The array methods that accept callback functions, such as sort(), can all benefit from simpler arrow function syntax, which changes seemingly complex processes into simpler code.

No arguments Binding

Arrow function does not contains arguments object as normal Functions

```
let myFunction=(x)=>console.log(arguments.length)
myFunction(3)  //error -> arguments is not defined
```

Expanded Object Functionality

Object Literal Syntax Extensions

The object literal is one of the most popular patterns in JavaScript. **JSON** is built on its syntax, and it's in nearly every JavaScript file on the Internet. The object literal's popularity is due to its succinct syntax for creating objects that would otherwise take several lines of code to create. Fortunately for developers, ECMAScript 6 makes object literals more powerful and even more succinct by extending the syntax in several ways.

Property Initializer Shorthand

```
let Title = "EcmaScript";
let version = "2015";
In ECMAscript 5 :
var Book = {Title:Title, version:version}
In ECMAScript 6:
var Book = {Title, version};
console.log(Book);  //output --> {Title: "EcmaScript", version: "2015"}
```

In ECMAScript 6, you can eliminate the duplication that exists around property names and local variables by using the property initializer shorthand syntax. When an object property name is the same as the local variable name, you can simply include the name without a colon and value.

When a property in an object literal only has a name, the JavaScript engine looks in the surrounding scope for a variable of the same name. If it finds one, that variable's value is assigned to the same name on the object literal

In ECMAscript 5 :

```
function createPerson(name, age) {
    return {
        name: name,
        age: age
    };
}
```

In ECMAscript 6 :

```
function createPerson(name, age) {
    return {  name, age};
}
createPerson("eman", 20)
```

Concise Methods

ECMAScript 6 also improves the syntax for assigning methods to object literals.

In **ECMAScript 5** and earlier, you must specify a name and then the full function definition to add a method to an object.

```
var person = {
    name: "Nicholas",
    sayName: function () {console.log(this.name);}
};
```

In **ECMAScript 6**, the syntax is made more concise by eliminating the colon and the function keyword.

```
var person = {
    name: "Nicholas",
    sayName() {console.log(this.name);}
};
```

Computed Property Names

ECMAScript 5 and earlier could compute property names on object instances when those properties were set with square brackets instead of dot notation. The square brackets allow you to specify property names using variables and string literals that might contain characters that would cause a syntax error if they were used in an identifier.

```
var person = {},
lastName = "last name";
person["first name"] = "Eman";
person[lastName] = "Fathi";
console.log(person["first name"]); //"Eman"
console.log(person[lastName]); //"Fathi"
```

Additionally, you can use string literals directly as property names in object literals

```
var person = {"first name": "Eman"};
console.log(person["first name"]); // "Eman"
```

In **ECMAScript 6**, computed property names are part of the object literal syntax, and they use the same square bracket notation that has been used to reference computed property names in object instances.

```
var titleParameter = "Title";
var titleValue = "ES6";


var Book = {
    [titleParameter]: titleValue,
    ["Book"+titleParameter]: titleValue,
    [titleValue+" printing"]: function () {
        return this[titleParameter] + " : " + this["Book" + titleParameter];
    }
};
 console.log(Book)//{Title: "ES6", BookTitle: "ES6", ES6 printing: ƒ}
```