Modern JS with ES6

A Brief History

A Brief History

- May 1995 - Originally developed under the name *Mocha*.

A Brief History

- May 1995 - Originally developed under the name *Mocha*.
- September 1995 - Released originally as *LiveScript*.

A Brief History

- May 1995 - Originally developed under the name *Mocha*.
- September 1995 - Released originally as *LiveScript*.
- December 1995 - Renamed *JavaScript* with the release of Netscape Navigator 2.0.

A Brief History

- May 1995 - Originally developed under the name *Mocha*.
- September 1995 - Released originally as *LiveScript*.
- December 1995 - Renamed *JavaScript* with the release of Netscape Navigator 2.0.
- November 1996 - Netscape submitted *JavaScript* to Ecma International for standardization.

A Brief History

- May 1995 - Originally developed under the name *Mocha*.
- September 1995 - Released originally as *LiveScript*.
- December 1995 - Renamed *JavaScript* with the release of Netscape Navigator 2.0.
- November 1996 - Netscape submitted *JavaScript* to Ecma International for standardization.
- June 1997 - This resulted in a new language standard, known as *ECMAScript*.

A Brief History

- May 1995 - Originally developed under the name *Mocha*.
- September 1995 - Released originally as *LiveScript*.
- December 1995 - Renamed *JavaScript* with the release of Netscape Navigator 2.0.
- November 1996 - Netscape submitted *JavaScript* to Ecma International for standardization.
- June 1997 - This resulted in a new language standard, known as *ECMAScript*.
- *ECMAScript* is the standard and *JavaScript* is the most popular implementation of that standard and also builds on top of it.

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998
- ES3: December 1999

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998
- ES3: December 1999
- ES4: Abandoned

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998
- ES3: December 1999
- ES4: Abandoned
- ES5: December 2009

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998
- ES3: December 1999
- ES4: Abandoned
- ES5: December 2009
- ES6/ES2015: June 2015

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998
- ES3: December 1999
- ES4: Abandoned
- ES5: December 2009
- ES6/ES2015: June 2015
- ES7/ES2016: June 2016

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998
- ES3: December 1999
- ES4: Abandoned
- ES5: December 2009
- ES6/ES2015: June 2015
- ES7/ES2016: June 2016
- ES8/ES2017: June 2017

ES_???

ES is short for ECMAScript. Every time you see ES followed by a number, it is referencing an edition of ECMAScript.

- ES1: June 1997
- ES2: June 1998
- ES3: December 1999
- ES4: Abandoned
- ES5: December 2009
- ES6/ES2015: June 2015
- ES7/ES2016: June 2016
- ES8/ES2017: June 2017
- ES.Next: This term is dynamic and references the next version of ECMAScript coming out.

TC39

The ECMA TC39 committee is responsible for evolving the ECMAScript programming language and authoring the specification.

TC39

The ECMA TC39 committee is responsible for evolving the ECMAScript programming language and authoring the specification.

- Stage 0 - Strawman

TC39

The ECMA TC39 committee is responsible for evolving the ECMAScript programming language and authoring the specification.

- Stage 0 - Strawman
- Stage 1 - Proposal

TC39

The ECMA TC39 committee is responsible for evolving the ECMAScript programming language and authoring the specification.

- Stage 0 - Strawman
- Stage 1 - Proposal
- Stage 2 - Draft

TC39

The ECMA TC39 committee is responsible for evolving the ECMAScript programming language and authoring the specification.

- Stage 0 - Strawman
- Stage 1 - Proposal
- Stage 2 - Draft
- Stage 3 - Candidate

ES6 Modules

Modules allow you to load code asynchronously and provides a layer of abstraction to your code.

ES6 Modules

Modules allow you to load code asynchronously and provides a layer of abstraction to your code.

Two ways to export from a module.

ES6 Modules

Modules allow you to load code asynchronously and provides a layer of abstraction to your code.

Two ways to export from a module.

- Multiple named exports

ES6 Modules

Modules allow you to load code asynchronously and provides a layer of abstraction to your code.

Two ways to export from a module.

- Multiple named exports
- Single default export

ES6 Modules - Multiple Named Exports

`mathlib.js`

ES6 Modules - Multiple Named Exports

mathlib.js

```
function square(x) {
    return x * x;
}
```

ES6 Modules - Multiple Named Exports

mathlib.js

```
function square(x) {
    return x * x;
}
function add(x, y) {
    return x + y;
}
```

ES6 Modules - Multiple Named Exports

mathlib.js

```
export function square(x) {
    return x * x;
}
export function add(x, y) {
    return x + y;
}
```

ES6 Modules - Multiple Named Exports

mathlib.js

```
export function square(x) {
    return x * x;
}
export function add(x, y) {
    return x + y;
}
```

main.js

ES6 Modules - Multiple Named Exports

mathlib.js

```
export function square(x) {
    return x * x;
}
export function add(x, y) {
    return x + y;
}
```

main.js

```
console.log(square(9)); // 81
console.log(add(4, 3)); // 7
```

ES6 Modules - Multiple Named Exports

mathlib.js

```
export function square(x) {
    return x * x;
}
export function add(x, y) {
    return x + y;
}
```

main.js

```
import { square, add } from 'mathlib';
console.log(square(9)); // 81
console.log(add(4, 3)); // 7
```

ES6 Modules - Single Default Exports

`foo.js`

ES6 Modules - Single Default Exports

---

foo.js

```
export default function() {
    console.log('Foo!');
}
```

ES6 Modules - Single Default Exports

foo.js

```
export default function() {
    console.log('Foo!');
}
```

main.js

```
import foo from 'foo';
foo(); // Foo!
```

ES6 Tools

BABEL

ES6 Tools

Variable Scoping

var vs let vs const

`var` is function scoped.

```
if ( true ) {
    var foo = 'bar';
}
console.log( foo );
// bar
```

`let` and `const` are block scoped.

```
if ( true ) {
    let foo = 'bar';
    const bar = 'foo';
}
console.log( foo );
console.log( bar );
// ReferenceError.
// ReferenceError.
```

let and const

```
let first = 'First string';
{
    let second = 'Second string';
    {
        let third = 'Third string';
    }
    // Accessing third here would throw a ReferenceError.
}
// Accessing second here would throw a ReferenceError.
// Accessing third here would throw a ReferenceError.
```

let and const

```
const first = 'First string';
{
    const second = 'Second string';
    {
        const third = 'Third string';
    }
    // Accessing third here would throw a ReferenceError.
}
// Accessing second here would throw a ReferenceError.
// Accessing third here would throw a ReferenceError.
```

const

`const` **variables can only be assigned once. It is <u>NOT</u> immutable.**

```
const foo = { bar: 1 };
foo = 'bar';
// “foo” is read only.
```

But, you can change the properties!

```
foo.bar = 2;
console.log(foo);
// { bar: 2 }
```

`Object.freeze()` **prevents changing the properties.**

```
const foo2 = Object.freeze(foo);
foo2.bar = 3;
console.log(foo2.bar); // 2
```

`Object.seal()` **prevents changing the object structure.**

```
Object.seal(foo);
foo.baz = false; // TypeError
```

Variable Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Variable Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Which means you can do this with ***functions*** and ***vars***:

```
sayHello();
function sayHello() {
    console.log('Hello!');
}
```

```
console.log( foobar );


var foobar = 'Woot!'
```

Variable Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Which means you can do this with ***functions*** and ***vars***:

```
sayHello(); // Hello!
function sayHello() {
    console.log('Hello!');
}
```

```
console.log( foobar );
// undefined

var foobar = 'Woot!'
```

Variable Hoisting

In ES6, `classes`, `let`, and `const` variables *are* hoisted but they are not initialized yet unlike `vars` and `functions`.

```
new Thing();
class Thing{};

console.log(foo);
let foo = true;

console.log(bar);
const bar = true;
```

Variable Hoisting

In ES6, `classes`, `let`, and `const` variables *are* hoisted but they are not initialized yet unlike `vars` and `functions`.

```
new Thing();
class Thing{};


console.log(foo);
let foo = true;


console.log(bar);
const bar = true;
```

```
// TypeError



// 'foo' was used before it was
defined



// 'bar' was used before it was
defined
```

Temporal Dead Zone

The variable is in a ***temporal dead zone*** from the start of the block until the initialization is processed.

```
if ( true ) { // TDZ starts!










}
```

Temporal Dead Zone

The variable is in a ***temporal dead zone*** from the start of the block until the initialization is processed.

```
if ( true ) { // TDZ starts!
    const doSomething = function () {
        console.log( thing ); // OK!
    };




}
```

Temporal Dead Zone

The variable is in a ***temporal dead zone*** from the start of the block until the initialization is processed.

```
if ( true ) { // TDZ starts!
    const doSomething = function () {
        console.log( thing ); // OK!
    };

    doSomething(); // ReferenceError



}
```

Temporal Dead Zone

The variable is in a ***temporal dead zone*** from the start of the block until the initialization is processed.

```
if ( true ) { // TDZ starts!
  const doSomething = function () {
    console.log( thing ); // OK!
  };

  doSomething(); // ReferenceError
  let thing = 'test'; // TDZ ends.


}
```

Temporal Dead Zone

The variable is in a ***temporal dead zone*** from the start of the block until the initialization is processed.

```
if ( true ) { // TDZ starts!
    const doSomething = function () {
        console.log( thing ); // OK!
    };

    doSomething(); // ReferenceError
    let thing = 'test'; // TDZ ends.
    doSomething();
    // Called outside TDZ!
}
```

But, what should I use?!? `var`? `let`? `const`?

The only difference between `const` and `let` is that `const` makes the contract that no rebinding will happen.

But, what should I use?!? `var`? `let`? `const`?

The only difference between `const` and `let` is that `const` makes the contract that no rebinding will happen.

Mathias Bynens - V8 Engineer @ Google

- Use `const` by default.
- Only use `let` if rebinding is needed.
- `var` shouldn’t be used in ES2015.

But, what should I use?!? `var`? `let`? `const`?

The only difference between `const` and `let` is that `const` makes the contract that no rebinding will happen.

Mathias Bynens - V8 Engineer @ Google

- Use `const` by default.
- Only use `let` if rebinding is needed.
- `var` shouldn't be used in ES2015.

Kyle Simpson - Founder @ Getify Solutions

- Use `var` for top level variables
- Use `let` for localized variables in smaller scopes.
- Refactor `let` to `const` only after some code has been written and you're reasonably sure there shouldn't be variable reassignment.

Iterables & Looping

When using `var`, you leak a global variable to the parent scope and the variable gets overwritten with every iteration.

Iterables & Looping

When using `var`, you leak a global variable to the parent scope and the variable gets overwritten with every iteration.

```
for ( var i = 0; i < 10; i++ ) {
    setTimeout( function() {
        console.log( 'Number: ' + i );
    }, 1000 );
}
```

Iterables & Looping

When using `var`, you leak a global variable to the parent scope and the variable gets overwritten with every iteration.

```
for ( var i = 0; i < 10; i++ ) {
    setTimeout( function() {
        console.log( 'Number: ' + i );
    }, 1000 );
}
```

```
// Number: 10          // Number: 10
// Number: 10          // Number: 10
// Number: 10          // Number: 10
// Number: 10          // Number: 10
// Number: 10          // Number: 10
```

Iterables & Looping

Using let in a for loop allows us to have the variable scoped to its block only.

Iterables & Looping

Using let in a for loop allows us to have the variable scoped to its block only.

```
for ( let i = 0; i < 10; i++ ) {
    setTimeout( function() {
        console.log( 'Number: ' + i );
    }, 1000 );
}
```

Iterables & Looping

Using let in a for loop allows us to have the variable scoped to its block only.

```
for ( let i = 0; i < 10; i++ ) {
    setTimeout( function() {
        console.log( 'Number: ' + i );
    }, 1000 );
}
```

```
// Number: 0        // Number: 5
// Number: 1        // Number: 6
// Number: 2        // Number: 7
// Number: 3        // Number: 8
// Number: 4        // Number: 9
```

Iterables & Looping

ES6 also gives us a new way to loop over iterables!

```
const iterable = [10, 20, 30];

for (const value of iterable) {
  console.log(value);
}


// 10

// 20

// 30
```

Iterables & Looping

ES6 also gives us a new way to loop over iterables!

```
const articleParagraphs = document.querySelectorAll('article > p');

for (const paragraph of articleParagraphs) {
  paragraph.classList.add('read');
}
```

Iterables & Looping

ES6 also gives us a new way to loop over iterables!

```
const foo = 'bar';

for (const letter of foo) {
  console.log(letter);
}


// b

// a

// r
```

Arrow Functions

Arrow Functions

More concise than traditional function expressions:

```
// Traditional function expression.
const addNumbers = function (num1, num2) {
   return num1 + num2;
}
```

Arrow Functions

More concise than traditional function expressions:

```
// Traditional function expression.
const addNumbers = function (num1, num2) {
   return num1 + num2;
}

// Arrow function expression.
const addNumbers = (num1, num2) => {
    return num1 + num2;
}
```

Arrow Functions

Arrow functions have implicit returns:

```javascript
// Traditional function expression.
const addNumbers = function (num1, num2) {
    return num1 + num2;
}

// Arrow function expression with implicit return.
const addNumbers = (num1, num2) => num1 + num2;
```

Arrow Functions

A few more examples:

```
// Arrow function without any arguments.
const sayHello = () => console.log( 'Hello!' );
```

Arrow Functions

A few more examples:

```
// Arrow function without any arguments.
const sayHello = () => console.log( 'Hello!' );

// Arrow function with a single argument.
const sayHello = name => console.log( `Hello ${name}!` );
```

Arrow Functions

A few more examples:

```
// Arrow function without any arguments.
const sayHello = () => console.log( 'Hello!' );

// Arrow function with a single argument.
const sayHello = name => console.log( `Hello ${name}!` );

// Arrow function with multiple arguments.
const sayHello = (fName, lName) => console.log( `Hello ${fName} ${lName}!` );
```

Arrow Functions

The value of `this` is picked up from its surroundings (lexical).

Therefore, you don't need `bind()`, `that`, or `self` anymore!

```
function Person(){
  this.age = 0;

  setInterval(function() {
    this.age++; // `this`refers to the Window. 😑
  }, 1000);
}
```

Arrow Functions

The value of `this` is picked up from its surroundings (lexical).

Therefore, you don't need `bind()`, `that`, or `self` anymore!

```
function Person(){
  var that = this;
  this.age = 0;

  setInterval(function() {
    that.age++; // Without arrow functions. Works, but is not ideal.
  }, 1000);
}
```

Arrow Functions

The value of `this` is picked up from its surroundings (lexical).

Therefore, you don't need `bind()`, `that`, or `self` anymore!

```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++; // `this` properly refers to the person object. 🎉🎉🎉
  }, 1000);
}
```

Arrow Functions

When should I not use arrow functions?

```
const button = document.querySelector('#my-button');
button.addEventListener('click', () => {
    this.classList.toggle('on');
})
```

Arrow Functions

When should I not use arrow functions?

```javascript
const button = document.querySelector('#my-button');
button.addEventListener('click', () => {
    this.classList.toggle('on'); // `this`refers to the Window. ☹
})
```

Default Arguments

Default Arguments

Here’s a basic function.

```
function calculateTotal( subtotal, tax, shipping ) {
  return subtotal + shipping + (subtotal * tax);
}

const total = calculateTotal(100, 0.07, 10);
```

Default Arguments

Let’s add some defaults to the arguments in our function expression!

```
function calculateTotal( subtotal, tax, shipping ) {
  return subtotal + shipping + (subtotal * tax);
}

const total = calculateTotal(100, 0.07, 10);
```

Default Arguments

The Old Way 😕

```
function calculateTotal( subtotal, tax, shipping ) {
  if ( tax === undefined ) {
    tax = 0.07;
  }
  if ( shipping === undefined ) {
    shipping = 10;
  }
  return subtotal + shipping + (subtotal * tax);
}

const total = calculateTotal(100);
```

Default Arguments

A little better?

```
function calculateTotal( subtotal, tax, shipping ) {
  tax = tax || 0.07;
  shipping = shipping || 10;

  return subtotal + shipping + (subtotal * tax);
}

const total = calculateTotal(100);
```

Default Arguments

Now with ES6! 🎉🎉🎉

```
function calculateTotal( subtotal, tax = 0.07, shipping = 10 ) {
  return subtotal + shipping + (subtotal * tax);
}


const total = calculateTotal(100);
```

Default Arguments

What if I wanted to only pass in the first and third argument?

```
function calculateTotal( subtotal, tax = 0.07, shipping = 10 ) {
  return subtotal + shipping + (subtotal * tax);
}


const total = calculateTotal(100, , 20); // Can I do this?
```

Default Arguments

What if I wanted to only pass in the first and third argument?

```
function calculateTotal( subtotal, tax = 0.07, shipping = 10 ) {
  return subtotal + shipping + (subtotal * tax);
}


const total = calculateTotal(100, , 20); // SyntaxError
```

Default Arguments

What if I wanted to only pass in the first and third argument?

```
function calculateTotal( subtotal, tax = 0.07, shipping = 10 ) {
  return subtotal + shipping + (subtotal * tax);
}

const total = calculateTotal(100, undefined, 20); // 🎉🎉🎉
```

Destructuring

Destructuring Objects

```
const person =  {
    first: 'Kevin',
    last: 'Langley',
    location: {
            city: 'Beverly Hills',
            state: 'Florida'
    }
};
```

Destructuring Objects

Let’s create some variables from the object properties.

```
const person =  {
    first: 'Kevin',
    last: 'Langley',
    location: {
            city: 'Beverly Hills',
            state: 'Florida'
    }
};

const first = person.first;
const last = person.last;
```

Destructuring Objects

Let’s do that using destructuring.

```
const person =  {
    first: 'Kevin',
    last: 'Langley',
    location: {
            city: 'Beverly Hills',
            state: 'Florida'
    }
};

const { first, last } = person;
```

Destructuring Objects

It even works with nested properties.

```
const person =  {
    first: 'Kevin',
    last: 'Langley',
    location: {
            city: 'Beverly Hills',
            state: 'Florida'
    }
};

const { first, last } = person;
const { city, state } = person.location;
```

Destructuring Objects

You can also rename the variables from the destructured object!

```
const person =  {
    first: 'Kevin',
    last: 'Langley',
    location: {
            city: 'Beverly Hills',
            state: 'Florida'
    }
};

const { first: fName, last: lName } = person;
const { city: locationCity, state: locationState } = person.location;
```

Destructuring Objects

What if I tried to destruct a property that doesn't exist?

```
const settings = { color: 'white', height: 500 };

const { width, height, color } = settings;
```

Destructuring Objects

What if I tried to destruct a property that doesn't exist?

```
const settings = { color: 'white', height: 500 };

const { width, height, color } = settings;

console.log(width); // undefined
console.log(height); // 500
console.log(color); // white
```

Destructuring Objects

But, you can set defaults in your destructuring!

```
const settings = { color: 'white', height: 500 };

const { width = 200, height = 200, color = 'black' } = settings;
```

Destructuring Objects

But, you can set defaults in your destructuring!

```
const settings = { color: 'white', height: 500 };

const { width = 200, height = 200, color = 'black' } = settings;

console.log(width); // 200
console.log(height); // 500
console.log(color); // white
```

Destructuring Arrays

You can destructure arrays as well!

```
const details = [ 'Kevin', 'Langley', 'kevinlangleyjr.com' ];
```

Destructuring Arrays

You can destructure arrays as well!

```
const details = [ 'Kevin', 'Langley', 'kevinlangleyjr.com' ];

const [ first, last, website ] = details;
```

Destructuring Arrays

You can destructure arrays as well!

```
const details = [ 'Kevin', 'Langley', 'kevinlangleyjr.com' ];

const [ first, last, website ] = details;

console.log(first); // Kevin
console.log(last); // Langley
console.log(website); // kevinlangleyjr.com
```

Spread... and ...Rest

...Spread Operator

Before ES6, we would run `.apply()` to pass in an array of arguments.

```
function doSomething (x, y, z) {
    console.log(x, y, z);
}
let args = [0, 1, 2];

// Call the function, passing args.
doSomething.apply(null, args);
```

...Spread Operator

But with ES6, we can use the spread operator `...` to pass in the arguments.

```
function doSomething (x, y, z) {
    console.log(x, y, z);
}
let args = [0, 1, 2];

// Call the function, without `apply`, passing args with the spread operator!
doSomething(...args);
```

...Spread Operator

We can also use the spread operator to combine arrays.

```
let array1 = ['one', 'two', 'three'];
let array2 = ['four', 'five'];


array1.push(...array2) // Adds array2 items to end of array
array1.unshift(...array2) //Adds array2 items to beginning of array
```

...Spread Operator

We can also use the spread operator to combine arrays at any point.

```
let array1 = ['two', 'three'];
let array2 = ['one', ...array1, 'four', 'five'];

console.log(array2); // ["one", "two", "three", "four", "five"]
```

...Spread Operator

We can also use the spread operator to create a copy of an array.

```
let array1 = [1,2,3];
let array2 = [...array1]; // like array1.slice()
array2.push(4)


console.log(array1); // [1,2,3]
console.log(array2); // [1,2,3,4]
```

...Spread Operator

We can also use the spread operator with destructuring.

```
const players = [ 'Kevin', 'Bobby', 'Nicole', 'Naomi', 'Jim', 'Sherry' ];

const [ first, second, third, ...unplaced ] = players;

console.log(first); // Kevin
console.log(second); // Bobby
console.log(third); // Nicole
console.log(unplaced); // ["Naomi", "Jim", "Sherry"]
```

...Spread Operator

We can also use the spread operator with destructuring.

```
const { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
console.log(x); // 1
console.log(y); // 2
console.log(z); // { a: 3, b: 4 }
```

...Spread Operator

We can also use the spread operator to expand a NodeList.

```
const elements = [...document.querySelectorAll('div')];

console.log(elements); // Lists all the div's on the page.
```

...Rest Operator

The rest operator allows us to more easily handle a variable number of function parameters.

```
function doMath(operator, ...numbers) {
  console.log(operator); // 'add'
  console.log(numbers); // [1, 2, 3]
}


doMath('add', 1, 2, 3);
```

Strings

Template Literals

The Template Literal, introduced in ES6, is a new way to create a string.

```
const name = 'Kevin';


// The old way...
console.log('Hello, ' + name + '!'); // Hello, Kevin!
```

Template Literals

The Template Literal, introduced in ES6, is a new way to create a string.

```
const name = 'Kevin';

// The old way...
console.log('Hello, ' + name + '!'); // Hello, Kevin!

// With ES6 template literals.
console.log(`Hello, ${name}!`); // Hello, Kevin!
```

Template Literals

Within template literals you can evaluate expressions.

```
const price = 19.99;
const tax = 0.07;


const total = `The total price is ${price + (price * tax)}`;
console.log(total);
// The total price is 21.3893
```

Template Literals

With template literals you can more easily create multi-line strings.

```
console.log('This is some text that flows across\ntwo lines!');

// "This is some text that flows across
// two lines!"

console.log(`But so does
this text!`);

// "But so does
// this text!"
```

New String Methods - `.startsWith()`

```
const str = 'Learn JavaScript Deeply';


console.log(str.startsWith('Learn'));       // true
console.log(str.startsWith('JavaScript')); // false
console.log(str.startsWith('Deeply', 17)); // true
```

New String Methods - `.endsWith()`

```
const str = 'Learn JavaScript Deeply';


console.log(str.endsWith('Deeply'));          // true
console.log(str.endsWith('Learn'));           // false
console.log(str.endsWith('JavaScript', 16)); // true
```

New String Methods - `.includes()`

```
const str = 'Learn JavaScript Deeply';


console.log(str.includes('JavaScript'));     // true
console.log(str.includes('Javascript'));     // false
console.log(str.includes('PHP'));            // false
```

New String Methods - `.repeat()`

```
const str = 'Deeply';


console.log(str.repeat(3));      // DeeplyDeeplyDeeply
console.log(str.repeat(2.5));    // DeeplyDeeply (converts to int)
console.log(str.repeat(-1));     // RangeError
```

Enhanced Object Literals

Enhanced Object Literals

```
const first = 'Kevin';
const last = 'Langley';
const age = 29;
```

Enhanced Object Literals

Let's assign our variables to properties of an object!

```
const first = 'Kevin';
const last = 'Langley';
const age = 29;

const person = {
    first: first,
    last: last,
    age: age
};
```

Enhanced Object Literals

Let's assign our variables to properties of an object!

```
const first = 'Kevin';
const last = 'Langley';
const age = 29;

const person = {
    first,
    last,
    age
};
```

Enhanced Object Literals

Let’s assign our variables to properties of an object!

```
const first = 'Kevin';
const last = 'Langley';
const age = 29;

const person = {
    firstName: first,
    lastName: last,
    age: age
};
```

Enhanced Object Literals

We can also use a shorter syntax for method definitions on objects initializers.

```
var obj = {
  foo: function() {
    console.log('foo');
  },
  bar: function() {
    console.log('bar');
  }
};
```

Enhanced Object Literals

We can also use a shorter syntax for method definitions on objects initializers.

```
const obj = {
  foo() {
    console.log('foo');
  },
  bar() {
    console.log('bar');
  }
};
```

Enhanced Object Literals

Or even define keys that evaluate on run time inside object literals.

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

console.log(a.foo1); // 1
console.log(a.foo2); // 2
console.log(a.foo3); // 3
```

Enhanced Object Literals

Let’s use template literals for those keys instead!

```
let i = 0;
const a = {
  [`foo${++i}`]: i,
  [`foo${++i}`]: i,     🎉🎉🎉
  [`foo${++i}`]: i
};

console.log(a.foo1); // 1
console.log(a.foo2); // 2
console.log(a.foo3); // 3
```

New Array Features!

Array.from()

```
const headers = document.querySelectorAll('h1');
```

Array.from()

```
const headers = document.querySelectorAll('h1');


const titles = headers.map(h1 => h1.textContent);
```

Array.from()

```
const headers = document.querySelectorAll('h1');

const titles = headers.map(h1 => h1.textContent);

// TypeError: headers.map is not a function
```

Array.from()

```javascript
const headers = document.querySelectorAll('h1');
const headersArray = [...headers];
const titles = headersArray.map(h1 => h1.textContent);
```

Array.from()

```
const headers = document.querySelectorAll('h1');
const headersArray = Array.from(headers);
const titles = headersArray.map(h1 => h1.textContent);
```

Array.from()

```
const headers = document.querySelectorAll('h1');
const titles = Array.from(headers, h1 => {
    return h1.textContent;
});
```

Array.from()

```
const titles = Array.from(document.querySelectorAll('h1'), h1 => {
    return h1.textContent;
});
```

Array.from()

```
const headers = Array.from(document.querySelectorAll('h1'));


const titles = headers.map(header => header.textContent);
```

Array.from()

```
const headers = document.querySelectorAll('h1');
const titles = Array.from(headers, header => header.textContent);
```

Array.of()

```
const values = Array.of(123, 456, 789);
```

Array.of()

```
const values = Array.of(123, 456, 789);
console.log(values);
```

Array.of()

```
const values = Array.of(123, 456, 789);
console.log(values); // [123,456,789]
```

Array.find()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];
```

Array.find()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];
```

Array.find()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];


 posts.2
```

Array.find()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];


 posts[2]
```

Array.find()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];

const post = posts.find(post => post.id === 2);
```

Array.find()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];

const post = posts.find(post => post.id === 2);

console.log(post);
```

Array.find()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];

const post = posts.find(post => post.id === 2);

console.log(post); // {id: 2, title: "Learn JS Deeply!"}
```

Array.findIndex()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];

const post = posts.findIndex(post => post.id === 2);
```

Array.findIndex()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];

const post = posts.findIndex(post => post.id === 2);

console.log(post);
```

Array.findIndex()

```
const posts = [
    {
        id: 1,
        title: 'Hello World!'
    },
    {
        id: 2,
        title: 'Learn JS Deeply!'
    }
];

const post = posts.findIndex(post => post.id === 2);

console.log(post); // 1
```

Promises

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

console.log(postsPromise);
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

console.log(postsPromise);

// Promise {<pending>}
// __proto__: Promise
// [[PromiseStatus]]: "pending"
// [[PromiseValue]]: undefinea
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise.then(data => console.log(data));
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise.then(data => console.log(data));

// Response {type: "cors", url: "https://2018.miami.wordcamp.org/wp-
json/wp/v2/posts", redirected: false, status: 200, ok: true, …}
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');


postsPromise.then(data => data.json())
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise.then(data => data.json()).then(data => console.log(data));
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise.then(data => data.json()).then(data => console.log(data));

// {id: 5060, date: "2018-03-15T17:41:09", ...}
// {id: 4954, date: "2018-03-14T00:21:10", ...}
// {id: 4943, date: "2018-03-13T19:16:11", ...}
// {id: 4702, date: "2018-03-10T11:04:36", ...}
// ...
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise
    .then(data => data.json())
    .then(data => console.log(data))
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise
    .then(data => data.json())
    .then(data => console.log(data))
    .catch(err);
```

Promises

```
const postsPromise = fetch('https://2018.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise
    .then(data => data.json())
    .then(data => console.log(data))
    .catch(err => console.error(err));
```

Promises

```
const postsPromise = fetch('https://2019.miami.wordcamp.org/wp-json/wp/v2/posts');


postsPromise
    .then(data => data.json())
    .then(data => console.log(data))
    .catch(err => console.error(err));
```

Promises

```
const postsPromise = fetch('https://2019.miami.wordcamp.org/wp-json/wp/v2/posts');

postsPromise
    .then(data => data.json())
    .then(data => console.log(data))
    .catch(err => console.error(err));

// TypeError: Failed to fetch
```

Promises

```
const p = new Promise((resolve, reject) => {


});
```

Promises

```
const p = new Promise((resolve, reject) => {
    resolve();
});
```

Promises

```
const p = new Promise((resolve, reject) => {
    reject();
});
```

Promises

```
const p = new Promise((resolve, reject) => {
    resolve('Learn JavaScript Deeply!');
});

p.then(data => console.log(data));
```

Promises

```
const p = new Promise((resolve, reject) => {
    resolve('Learn JavaScript Deeply!');
});

p.then(data => console.log(data)); // Learn JavaScript Deeply
```

Promises

```
const p = new Promise((resolve, reject) => {
    reject(Error('Uh oh!'));
});

p.then(data => console.log(data));
```

Promises

```
const p = new Promise((resolve, reject) => {
    reject(Error('Uh oh!'));
});

p.then(data => console.log(data));

// Uncaught (in promise) Error: Uh oh!
```

Promises

```
const p = new Promise((resolve, reject) => {
    reject(Error('Uh oh!'));
});

p
    .then(data => console.log(data));
    .catch(err => console.error(err));

// Error: Uh oh!
```

Classes

Classes

```
// Class declaration
class Animal {

}

// Class expression
const Animal = class {


}
```

Classes

```
class Animal {




}
```

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }



}
```

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}
```

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {



}
```

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks!`);
  }
}
```

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks!`);
  }
}

const puppy = new Dog('Spot');
puppy.speak(); // Spot barks!
```

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    this.breed = breed;
  }
  speak() {
    console.log(`${this.name} barks!`);
  }
}
```

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
  speak() {
    console.log(`${this.name} barks!`);
  }
}
```

Async & Await

```
function resolveAfter(time) {






}
```

Async & Await

```
function resolveAfter(time) {
  return new Promise(resolve => {



  });
}
```

Async & Await

```
function resolveAfter(time) {
  return new Promise(resolve => {
    setTimeout(() => {

    }, time);
  });
}
```

Async & Await

```
function resolveAfter(time) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Resolved after ${time} milliseconds');
    }, time);
  });
}
```

Async & Await

```javascript
function resolveAfter(time) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Resolved after ${time} milliseconds');
    }, time);
  });
}

console.log('Starting resolveAfter()');

const result = resolveAfter(500);
console.log(result);

console.log('Ending resolveAfter()');
```

```
// Starting resolveAfter()
// Promise {<pending>}
// __proto__: Promise
// [[PromiseStatus]]: "pending"
// [[PromiseValue]]: undefined
// Ending resolveAfter()
```

Async & Await

```
async function resolveAfter(time) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Resolved after ${time} milliseconds');
    }, time);
  });
}

console.log('Starting resolveAfter()');

const result = await resolveAfter(500);
console.log(result);

console.log('Ending resolveAfter()');
```

```
// SyntaxError: await is only valid in async
function
```

Async & Await

```
function resolveAfter(time) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Resolved after ${time} milliseconds');
    }, time);
  });
}

async function asyncCall() {
  console.log('Starting asyncCall()');

  const result = await resolveAfter(500);
  console.log(result);

  console.log('Ending asyncCall()');
}

asyncCall();
```

```
// 08:14:22.852 Starting asyncCall()
// 08:14:23.474 Resolved after 500 milliseconds
// 08:14:38.483 Ending asyncCall()
```

Async & Await

```
function resolveAfter(time) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Resolved after ${time} milliseconds');
    }, time);
  });
}

async function asyncCall() {
  console.log('Starting asyncCall()');

  const result1 = await resolveAfter(500);   // 08:14:22.852 Starting asyncCall()
  console.log(result1);                      // 08:14:23.474 Resolved after 500 milliseconds
                                             // 08:14:28.478 Resolved after 5000 milliseconds
  const result2 = await resolveAfter(5000);  // 08:14:38.483 Ending asyncCall()
  console.log(result2);

  console.log('Ending asyncCall()');
}

asyncCall();
```

Features! Features! Features!

- Generators

Features! Features! Features!

- Generators
- Symbols

Features! Features! Features!

- Generators
- Symbols
- Proxies

Features! Features! Features!

- Generators
- Symbols
- Proxies
- Sets and WeakSets

Features! Features! Features!

- Generators
- Symbols
- Proxies
- Sets and WeakSets
- Maps and WeakMaps

Features! Features! Features!

- Generators
- Symbols
- Proxies
- Sets and WeakSets
- Maps and WeakMaps
- And even more that is still in review by TC39!

Still have questions?

Wes Bos - ES6 for Everyone - http://es6.io

Thank you!