

# Learn ES2015

A detailed overview of ECMAScript 2015 features. Based on Luke Hoban's `es6features` repo.

[Edit this page \(https://github.com/babel/babel.github.io/blob/master/learn-es2015.md\)](https://github.com/babel/babel.github.io/blob/master/learn-es2015.md)

## es6features

This document was originally taken from Luke Hoban's excellent `es6features` (<https://git.io/es6features>) repo. Go give it a star on GitHub!

## REPL

Be sure to try these features out in the online REPL ([/repl](https://babeljs.io/repl)).

## Introduction

ECMAScript 2015 is an ECMAScript standard that was ratified in June 2015.

ES2015 is a significant update to the language, and the first major update to the language since ES5 was standardized in 2009. Implementation of these features in major JavaScript engines is underway now (<https://kangax.github.io/es5-compat-table/es6/>).

See the ES2015 standard (<http://www.ecma-international.org/ecma-262/6.0/index.html>) for full specification of the ECMAScript 2015 language.

## ECMAScript 2015 Features

### Arrows and Lexical This

Arrows are a function shorthand using the `=>` syntax. They are syntactically similar to the related feature in C#, Java 8 and CoffeeScript. They support both expression and statement bodies. Unlike functions, arrows share the same lexical `this` as their surrounding code. If an arrow is inside another function, it shares the “arguments” variable of its parent function.

JavaScript

```
// Expression bodies
var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);

// Statement bodies
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});

// Lexical this
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
};

// Lexical arguments
function square() {
  let example = () => {
    let numbers = [];
    for (let number of arguments) {
      numbers.push(number * number);
    }

    return numbers;
  };

  return example();
}

square(2, 4, 7.5, 8, 11.5, 21); // returns: [4, 16, 56.25, 64, 132.25, 441]
```

## Classes

ES2015 classes are a simple sugar over the prototype-based OO pattern. Having a single convenient declarative form makes class patterns easier to use, and encourages interoperability. Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

JavaScript

```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);

    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
    //...
  }
  update(camera) {
    //...
    super.update();
  }
  static defaultMatrix() {
    return new THREE.Matrix4();
  }
}
```

## Enhanced Object Literals

Object literals are extended to support setting the prototype at construction, shorthand for `foo: foo` assignments, defining methods and making super calls. Together, these also bring object literals and class declarations closer together, and let object-based design benefit from some of the same conveniences.

JavaScript

```
var obj = {
  // Sets the prototype. "__proto__" or '__proto__' would also work.
  __proto__: theProtoObj,
  // Computed property name does not set prototype or trigger early error for
  // duplicate __proto__ properties.
  ['__proto__']: somethingElse,
  // Shorthand for 'handler: handler'
  handler,
  // Methods
  toString() {
    // Super calls
    return "d " + super.toString();
  },
  // Computed (dynamic) property names
  [ "prop_" + (() => 42)() ]: 42
};
```

The `__proto__` property requires native support, and was deprecated in previous ECMAScript versions. Most engines now support the property, but some do not ([https://kangax.github.io/compat-table/es6/#\\_\\_proto\\_\\_in\\_object\\_literals](https://kangax.github.io/compat-table/es6/#__proto__in_object_literals)). Also, note that only web browsers (<http://www.ecma-international.org/ecma-262/6.0/index.html#sec-additional-ecmascript-features-for-web-browsers>) are required to implement it, as it's in Annex B ([http://www.ecma-international.org/ecma-262/6.0/index.html#sec-object.prototype.\\_\\_proto\\_\\_](http://www.ecma-international.org/ecma-262/6.0/index.html#sec-object.prototype.__proto__)). It is available in Node.

## Template Strings

Template strings provide syntactic sugar for constructing strings. This is similar to string interpolation features in Perl, Python and more. Optionally, a tag can be added to allow the string construction to be customized, avoiding injection attacks or constructing higher level data structures from string contents.

### JavaScript

```
// Basic literal string creation
`This is a pretty little template string.`

// Multiline strings
`In ES5 this is
not legal.`

// Interpolate variable bindings
var name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`

// Unescaped template strings
String.raw`In ES5 "\n" is a line-feed.`

// Construct an HTTP request prefix is used to interpret the replacements and construction
GET`http://foo.org/bar?a=${a}&b=${b}
Content-Type: application/json
X-Credentials: ${credentials}
{ "foo": ${foo},
  "bar": ${bar}}`(myOnReadyStateChangeHandler);
```

## Destructuring

Destructuring allows binding using pattern matching, with support for matching arrays and objects. Destructuring is fail-soft, similar to standard object lookup `foo["bar"]`, producing `undefined` values when not found.

### JavaScript

```
// list matching
var [a, ,b] = [1,2,3];
a === 1;
b === 3;

// object matching
var { op: a, lhs: { op: b }, rhs: c }
  = getASTNode()

// object matching shorthand
// binds `op`, `lhs` and `rhs` in scope
var {op, lhs, rhs} = getASTNode()

// Can be used in parameter position
function g({name: x}) {
  console.log(x);
}
g({name: 5})

// Fail-soft destructuring
var [a] = [];
a === undefined;

// Fail-soft destructuring with defaults
var [a = 1] = [];
a === 1;

// Destructuring + defaults arguments
function r({x, y, w = 10, h = 10}) {
  return x + y + w + h;
}
r({x:1, y:2}) === 23
```

## Default + Rest + Spread

Callee-evaluated default parameter values. Turn an array into consecutive arguments in a function call. Bind trailing parameters to an array. Rest replaces the need for arguments and addresses common cases more directly.

JavaScript

```
function f(x, y=12) {
  // y is 12 if not passed (or passed as undefined)
  return x + y;
}
f(3) == 15
```

JavaScript

```
function f(x, ...y) {
  // y is an Array
  return x * y.length;
}
f(3, "hello", true) == 6
```

JavaScript

```
function f(x, y, z) {
  return x + y + z;
}
// Pass each elem of array as argument
f(...[1,2,3]) == 6
```

## Let + Const

Block-scoped binding constructs. `let` is the new `var`. `const` is single-assignment. Static restrictions prevent use before assignment.

JavaScript

```
function f() {
  {
    let x;
    {
      // this is ok since it's a block scoped name
      const x = "sneaky";
      // error, was just defined with `const` above
      x = "foo";
    }
    // this is ok since it was declared with `let`
    x = "bar";
    // error, already declared above in this block
    let x = "inner";
  }
}
```

## Iterators + For..Of

Iterator objects enable custom iteration like CLR `IEnumerable` or Java `Iterable`. Generalize `for...in` to custom iterator-based iteration with `for...of`. Don't require realizing an array, enabling lazy design patterns like LINQ.

### JavaScript

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur };
      }
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Iteration is based on these duck-typed interfaces (using TypeScript (<http://typescriptlang.org>) type syntax for exposition only):

### TypeScript

```
interface IteratorResult {
  done: boolean;
  value: any;
}
interface Iterator {
  next(): IteratorResult;
}
interface Iterable {
  [Symbol.iterator](): Iterator
}
```

### Support via polyfill

In order to use Iterators you must include the Babel polyfill (</docs/usage/polyfill>).

## Generators

Generators simplify iterator-authoring using `function*` and `yield`. A function declared as `function*` returns a Generator instance. Generators are subtypes of iterators which include additional `next` and `throw`. These enable values to flow back into the generator, so `yield` is an expression form which returns a value (or throws).

Note: Can also be used to enable 'await'-like async programming, see also ES7 `await` proposal (<https://github.com/lukehoban/ecmascript-asyncawait>).

### JavaScript

```

var fibonacci = {
  [Symbol.iterator]: function*() {
    var pre = 0, cur = 1;
    for (;;) {
      var temp = pre;
      pre = cur;
      cur += temp;
      yield cur;
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}

```

The generator interface is (using TypeScript (<http://typescriptlang.org>) type syntax for exposition only):

#### TypeScript

```

interface Generator extends Iterator {
  next(value?: any): IteratorResult;
  throw(exception: any);
}

```

### Support via polyfill

In order to use Generators you must include the Babel polyfill (</docs/usage/polyfill>).

## Comprehensions

Removed in Babel 6.0

## Unicode

Non-breaking additions to support full Unicode, including new unicode literal form in strings and new RegExp `u` mode to handle code points, as well as new APIs to process strings at the 21bit code points level. These additions support building global apps in JavaScript.

#### JavaScript

```

// same as ES5.1
"吉".length == 2

// new RegExp behaviour, opt-in 'u'
"吉".match(/./u)[0].length == 2

// new form
"\u{20BB7}" == "吉" == "\uD842\uDFB7"

// new String ops
"吉".codePointAt(0) == 0x20BB7

// for-of iterates code points
for(var c of "吉") {
  console.log(c);
}

```

## Modules

Language-level support for modules for component definition. Codifies patterns from popular JavaScript module loaders (AMD, CommonJS). Runtime behaviour defined by a host-defined default loader. Implicitly async model – no code executes until requested modules are available and processed.

#### JavaScript

```

// lib/math.js
export function sum(x, y) {
  return x + y;
}

export var pi = 3.141593;

```

#### JavaScript

```
// app.js
import * as math from "lib/math";
console.log("2π = " + math.sum(math.pi, math.pi));
```

#### JavaScript

```
// otherApp.js
import {sum, pi} from "lib/math";
console.log("2π = " + sum(pi, pi));
```

Some additional features include `export default` and `export *`:

#### JavaScript

```
// lib/mathplusplus.js
export * from "lib/math";
export var e = 2.71828182846;
export default function(x) {
  return Math.exp(x);
}
```

#### JavaScript

```
// app.js
import exp, {pi, e} from "lib/mathplusplus";
console.log("e^π = " + exp(pi));
```

## Module Formatters

Babel can transpile ES2015 Modules to several different formats including Common.js, AMD, System, and UMD. You can even create your own. For more details see the modules docs (</docs/usage/modules>).

## Module Loaders

### Not part of ES2015

This is left as implementation-defined within the ECMAScript 2015 specification. The eventual standard will be in WHATWG's Loader specification (<https://whatwg.github.io/loader/>), but that is currently a work in progress. What is below is from a previous ES2015 draft.

Module loaders support:

- Dynamic loading
- State isolation
- Global namespace isolation
- Compilation hooks
- Nested virtualization

The default module loader can be configured, and new loaders can be constructed to evaluate and load code in isolated or constrained contexts.

#### JavaScript

```
// Dynamic loading - 'System' is default loader
System.import("lib/math").then(function(m) {
  alert("2π = " + m.sum(m.pi, m.pi));
});

// Create execution sandboxes - new Loaders
var loader = new Loader({
  global: fixup(window) // replace 'console.log'
});
loader.eval("console.log(\"hello world!\");");

// Directly manipulate module cache
System.get("jquery");
System.set("jquery", Module({$: $})); // WARNING: not yet finalized
```

### Additional polyfill needed

Since Babel defaults to using common.js modules, it does not include the polyfill for the module loader API. Get it here (<https://github.com/ModuleLoader/es6-module-loader>).

## Using Module Loader

In order to use this, you'll need to tell Babel to use the `system` module formatter. Also be sure to check out System.js (<https://github.com/systemjs/systemjs>)

## Map + Set + WeakMap + WeakSet

Efficient data structures for common algorithms. WeakMaps provides leak-free object-key'd side tables.

### JavaScript

```
// Sets
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;

// Maps
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;

// Weak Maps
var wm = new WeakMap();
wm.set(s, { extra: 42 });
wm.size === undefined

// Weak Sets
var ws = new WeakSet();
ws.add({ data: 42 });
// Because the added object has no other references, it will not be held in the set
```

## Support via polyfill

In order to support Maps, Sets, WeakMaps, and WeakSets in all environments you must include the Babel polyfill (</docs/usage/polyfill>).

## Proxies

Proxies enable creation of objects with the full range of behaviors available to host objects. Can be used for interception, object virtualization, logging/profiling, etc.

### JavaScript

```
// Proxying a normal object
var target = {};
var handler = {
  get: function (receiver, name) {
    return `Hello, ${name}!`;
  }
};

var p = new Proxy(target, handler);
p.world === "Hello, world!";
```

### JavaScript

```
// Proxying a function object
var target = function () { return "I am the target"; };
var handler = {
  apply: function (receiver, ...args) {
    return "I am the proxy";
  }
};

var p = new Proxy(target, handler);
p() === "I am the proxy";
```

There are traps available for all of the runtime-level meta-operations:

### JavaScript



```

var handler =
{
  // target.prop
  get: ...,
  // target.prop = value
  set: ...,
  // 'prop' in target
  has: ...,
  // delete target.prop
  deleteProperty: ...,
  // target(...args)
  apply: ...,
  // new target(...args)
  construct: ...,
  // Object.getOwnPropertyDescriptor(target, 'prop')
  getOwnPropertyDescriptor: ...,
  // Object.defineProperty(target, 'prop', descriptor)
  defineProperty: ...,
  // Object.getPrototypeOf(target), Reflect.getPrototypeOf(target),
  // target.__proto__, Object.isPrototypeOf(target), Object.getPrototypeOf(target)
  getPrototypeOf: ...,
  // Object.setPrototypeOf(target), Reflect.setPrototypeOf(target)
  setPrototypeOf: ...,
  // for (let i in target) {}
  enumerate: ...,
  // Object.keys(target)
  ownKeys: ...,
  // Object.preventExtensions(target)
  preventExtensions: ...,
  // Object.isExtensible(target)
  isExtensible :...
}

```

## Unsupported feature

Due to the limitations of ES5, Proxies cannot be transpiled or polyfilled. See support in various JavaScript engines (<https://kangax.github.io/compat-table/es6/#Proxy>).

## Symbols

Symbols enable access control for object state. Symbols allow properties to be keyed by either `string` (as in ES5) or `symbol`. Symbols are a new primitive type. Optional `name` parameter used in debugging - but is not part of identity. Symbols are unique (like `gensym`), but not private since they are exposed via reflection features like `Object.getOwnPropertySymbols`.

### JavaScript

```

(function() {

  // module scoped symbol
  var key = Symbol("key");

  function MyClass(privateData) {
    this[key] = privateData;
  }

  MyClass.prototype = {
    doStuff: function() {
      ... this[key] ...
    }
  };

  // Limited support from Babel, full support requires native implementation.
  typeof key === "symbol"
})();

var c = new MyClass("hello")
c["key"] === undefined

```

## Limited support via polyfill

Limited support requires the Babel polyfill (</docs/usage/polyfill>). Due to language limitations, some features can't be transpiled or polyfilled. See `core.js`'s caveats section (<https://github.com/zloirock/core-js#caveats-when-using-symbol-polyfill>) for more details.

## Subclassable Built-ins

In ES2015, built-ins like `Array`, `Date` and `DOM Element`s can be subclassed.

#### JavaScript

```
// User code of Array subclass
class MyArray extends Array {
  constructor(...args) { super(...args); }
}

var arr = new MyArray();
arr[1] = 12;
arr.length == 2
```

### Partial support

Built-in subclassability should be evaluated on a case-by-case basis as classes such as `HTMLElement` **can** be subclassed while many such as `Date`, `Array` and `Error` **cannot** be due to ES5 engine limitations.

## Math + Number + String + Object APIs

Many new library additions, including core Math libraries, Array conversion helpers, and `Object.assign` for copying.

#### JavaScript

```
Number.EPSILON
Number.isInteger(Infinity) // false
Number.isNaN("NaN") // false

Math.acosh(3) // 1.762747174039086
Math.hypot(3, 4) // 5
Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

"abcde".includes("cd") // true
"abc".repeat(3) // "abcabcabc"

Array.from(document.querySelectorAll("*")) // Returns a real Array
Array.of(1, 2, 3) // Similar to new Array(...), but without special one-arg behavior
[0, 0, 0].fill(7, 1) // [0,7,7]
[1,2,3].findIndex(x => x == 2) // 1
["a", "b", "c"].entries() // iterator [0, "a"], [1,"b"], [2,"c"]
["a", "b", "c"].keys() // iterator 0, 1, 2
["a", "b", "c"].values() // iterator "a", "b", "c"

Object.assign(Point, { origin: new Point(0,0) })
```

### Limited support from polyfill

Most of these APIs are supported by the Babel polyfill (</docs/usage/polyfill>). However, certain features are omitted for various reasons (e.g. `String.prototype.normalize` needs a lot of additional code to support). You can find more polyfills here (<https://github.com/addyosmani/es6-tools#polyfills>).

## Binary and Octal Literals

Two new numeric literal forms are added for binary ( `b` ) and octal ( `o` ).

#### JavaScript

```
0b111110111 === 503 // true
0o767 === 503 // true
```

### Only supports literal form

Babel is only able to transform `0o767` and not `Number("0o767")`.

## Promises

Promises are a library for asynchronous programming. Promises are a first class representation of a value that may be made available in the future. Promises are used in many existing JavaScript libraries.

#### JavaScript

```
function timeout(duration = 0) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, duration);
  })
}

var p = timeout(1000).then(() => {
  return timeout(2000);
}).then(() => {
  throw new Error("hmm");
}).catch(err => {
  return Promise.all([timeout(100), timeout(200)]);
})
```

### Support via polyfill

In order to support Promises you must include the Babel polyfill (</docs/usage/polyfill>).

## Reflect API

Full reflection API exposing the runtime-level meta-operations on objects. This is effectively the inverse of the Proxy API, and allows making calls corresponding to the same meta-operations as the proxy traps. Especially useful for implementing proxies.

### JavaScript

```
var O = {a: 1};
Object.defineProperty(O, 'b', {value: 2});
O[Symbol('c')] = 3;

Reflect.ownKeys(O); // ['a', 'b', Symbol(c)]

function C(a, b){
  this.c = a + b;
}
var instance = Reflect.construct(C, [20, 22]);
instance.c; // 42
```

### Support via polyfill

In order to use the Reflect API you must include the Babel polyfill (</docs/usage/polyfill>).

## Tail Calls

Calls in tail-position are guaranteed to not grow the stack unboundedly. Makes recursive algorithms safe in the face of unbounded inputs.

### JavaScript

```
function factorial(n, acc = 1) {
  "use strict";
  if (n <= 1) return acc;
  return factorial(n - 1, n * acc);
}

// Stack overflow in most implementations today,
// but safe on arbitrary inputs in ES2015
factorial(100000)
```

### Temporarily Removed in Babel 6

Only explicit self referencing tail recursion was supported due to the complexity and performance impact of supporting tail calls globally. Removed due to other bugs and will be re-implemented.

## Community Discussion

[Start Discussion](#) 0 replies

Babel (<https://github.com/babel/babel>) · Distributed under MIT License (<https://github.com/babel/babel/blob/master/LICENSE>) · Code of Conduct

([https://github.com/babel/babel/blob/master/CODE\\_OF\\_CONDUCT.md](https://github.com/babel/babel/blob/master/CODE_OF_CONDUCT.md))

Looking for Babel 5.x docs? (<http://henryzoo.com/babel.github.io/>) · Found an issue with the docs? Report it here (<https://github.com/babel/babel.github.io/issues/new>).