**ECMAScript 6**[**git.io/es6features**](http://git.io/es6features)

**Introduction**

ECMAScript 6, also known as ECMAScript 2015, is the latest version of the ECMAScript standard. ES6 is a significant update to the language, and the first update to the language since ES5 was standardized in 2009. Implementation of these features in major JavaScript engines is [underway now](http://kangax.github.io/es5-compat-table/es6/).

See the [ES6 standard](http://www.ecma-international.org/ecma-262/6.0/) for full specification of the ECMAScript 6 language.

ES6 includes the following new features:

* [arrows](https://github.com/lukehoban/es6features#arrows)
* [classes](https://github.com/lukehoban/es6features#classes)
* [enhanced object literals](https://github.com/lukehoban/es6features#enhanced-object-literals)
* [template strings](https://github.com/lukehoban/es6features#template-strings)
* [destructuring](https://github.com/lukehoban/es6features#destructuring)
* [default + rest + spread](https://github.com/lukehoban/es6features#default--rest--spread)
* [let + const](https://github.com/lukehoban/es6features#let--const)
* [iterators + for..of](https://github.com/lukehoban/es6features#iterators--forof)
* [generators](https://github.com/lukehoban/es6features#generators)
* [unicode](https://github.com/lukehoban/es6features#unicode)
* [modules](https://github.com/lukehoban/es6features#modules)
* [module loaders](https://github.com/lukehoban/es6features#module-loaders)
* [map + set + weakmap + weakset](https://github.com/lukehoban/es6features#map--set--weakmap--weakset)
* [proxies](https://github.com/lukehoban/es6features#proxies)
* [symbols](https://github.com/lukehoban/es6features#symbols)
* [subclassable built-ins](https://github.com/lukehoban/es6features#subclassable-built-ins)
* [promises](https://github.com/lukehoban/es6features#promises)
* [math + number + string + array + object APIs](https://github.com/lukehoban/es6features#math--number--string--array--object-apis)
* [binary and octal literals](https://github.com/lukehoban/es6features#binary-and-octal-literals)
* [reflect api](https://github.com/lukehoban/es6features#reflect-api)
* [tail calls](https://github.com/lukehoban/es6features#tail-calls)

**ECMAScript 6 Features**

**Arrows**

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 statement block bodies as well as expression bodies which return the value of the expression. Unlike functions, arrows share the same lexical this as their surrounding code.

// Expression bodies

var odds = evens.map(v => v + 1);

var nums = evens.map((v, i) => v + i);

var pairs = evens.map(v => ({even: v, odd: v + 1}));

// 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));

}

}

More info: [MDN Arrow Functions](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

**Classes**

ES6 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.

class SkinnedMesh extends THREE.Mesh {

constructor(geometry, materials) {

super(geometry, materials);

this.idMatrix = SkinnedMesh.defaultMatrix();

this.bones = [];

this.boneMatrices = [];

//...

}

update(camera) {

//...

super.update();

}

get boneCount() {

return this.bones.length;

}

set matrixType(matrixType) {

this.idMatrix = SkinnedMesh[matrixType]();

}

static defaultMatrix() {

return new THREE.Matrix4();

}

}

More info: [MDN Classes](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes)

**Enhanced Object Literals**

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

var obj = {

// \_\_proto\_\_

\_\_proto\_\_: theProtoObj,

// Shorthand for ‘handler: handler’

handler,

// Methods

toString() {

// Super calls

return "d " + super.toString();

},

// Computed (dynamic) property names

[ 'prop\_' + (() => 42)() ]: 42

};

More info: [MDN Grammer and types: Object literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Object_literals)

**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.

// Basic literal string creation

`In JavaScript '\n' is a line-feed.`

// Multiline strings

`In JavaScript this is

not legal.`

// String interpolation

var name = "Bob", time = "today";

`Hello ${name}, how are you ${time}?`

// Construct an HTTP request prefix is used to interpret the replacements and construction

POST`http://foo.org/bar?a=${a}&b=${b}

Content-Type: application/json

X-Credentials: ${credentials}

{ "foo": ${foo},

"bar": ${bar}}`(myOnReadyStateChangeHandler);

More info: [MDN Template Strings](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/template_strings)

**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.

// list matching

var [a, , b] = [1,2,3];

// 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;

More info: [MDN Destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

**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.

function f(x, y=12) {

// y is 12 if not passed (or passed as undefined)

return x + y;

}

f(3) == 15

function f(x, ...y) {

// y is an Array

return x \* y.length;

}

f(3, "hello", true) == 6

function f(x, y, z) {

return x + y + z;

}

// Pass each elem of array as argument

f(...[1,2,3]) == 6

More MDN info: [Default parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters), [Rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters), [Spread Operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

**Let + Const**

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

function f() {

{

let x;

{

// okay, block scoped name

const x = "sneaky";

// error, const

x = "foo";

}

// error, already declared in block

let x = "inner";

}

}

More MDN info: [let statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), [const statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const)

**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.

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):

interface IteratorResult {

done: boolean;

value: any;

}

interface Iterator {

next(): IteratorResult;

}

interface Iterable {

[Symbol.iterator](): Iterator

}

More info: [MDN for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of)

**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.

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):

interface Generator extends Iterator {

next(value?: any): IteratorResult;

throw(exception: any);

}

More info: [MDN Iteration protocols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols)

**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.

// same as ES5.1

"𠮷".length == 2

// new RegExp behaviour, opt-in ‘u’

"𠮷".match(/./u)[0].length == 2

// new form

"\u{20BB7}"=="𠮷"=="\uD842\uDFB7"

// new String ops

"𠮷".codePointAt(0) == 0x20BB7

// for-of iterates code points

for(var c of "𠮷") {

console.log(c);

}

More info: [MDN RegExp.prototype.unicode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/unicode)

**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.

// lib/math.js

export function sum(x, y) {

return x + y;

}

export var pi = 3.141593;

// app.js

import \* as math from "lib/math";

alert("2π = " + math.sum(math.pi, math.pi));

// otherApp.js

import {sum, pi} from "lib/math";

alert("2π = " + sum(pi, pi));

Some additional features include export default and export \*:

// lib/mathplusplus.js

export \* from "lib/math";

export var e = 2.71828182846;

export default function(x) {

return Math.log(x);

}

// app.js

import ln, {pi, e} from "lib/mathplusplus";

alert("2π = " + ln(e)\*pi\*2);

More MDN info: [import statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import), [export statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export)

**Module Loaders**

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.

// 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

**Map + Set + WeakMap + WeakSet**

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

// 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

More MDN info: [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), [WeakMap](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap), [WeakSet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet)

**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.

// 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!';

// 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:

var handler =

{

get:...,

set:...,

has:...,

deleteProperty:...,

apply:...,

construct:...,

getOwnPropertyDescriptor:...,

defineProperty:...,

getPrototypeOf:...,

setPrototypeOf:...,

enumerate:...,

ownKeys:...,

preventExtensions:...,

isExtensible:...

}

More info: [MDN Proxy](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy)

**Symbols**

Symbols enable access control for object state. Symbols allow properties to be keyed by either string (as in ES5) orsymbol. Symbols are a new primitive type. Optional description 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 likeObject.getOwnPropertySymbols.

var MyClass = (function() {

// module scoped symbol

var key = Symbol("key");

function MyClass(privateData) {

this[key] = privateData;

}

MyClass.prototype = {

doStuff: function() {

... this[key] ...

}

};

return MyClass;

})();

var c = new MyClass("hello")

c["key"] === undefined

More info: [MDN Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)

**Subclassable Built-ins**

In ES6, built-ins like Array, Date and DOM Elements can be subclassed.

Object construction for a function named Ctor now uses two-phases (both virtually dispatched):

* Call Ctor[@@create] to allocate the object, installing any special behavior
* Invoke constructor on new instance to initialize

The known @@create symbol is available via Symbol.create. Built-ins now expose their @@create explicitly.

// Pseudo-code of Array

class Array {

constructor(...args) { /\* ... \*/ }

static [Symbol.create]() {

// Install special [[DefineOwnProperty]]

// to magically update 'length'

}

}

// User code of Array subclass

class MyArray extends Array {

constructor(...args) { super(...args); }

}

// Two-phase 'new':

// 1) Call @@create to allocate object

// 2) Invoke constructor on new instance

var arr = new MyArray();

arr[1] = 12;

arr.length == 2

**Math + Number + String + Array + Object APIs**

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

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].find(x => x == 3) // 3

[1, 2, 3].findIndex(x => x == 2) // 1

[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]

["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) })

More MDN info: [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number), [Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math), [Array.from](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from), [Array.of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/of), [Array.prototype.copyWithin](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/copyWithin), [Object.assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)

**Binary and Octal Literals**

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

0b111110111 === 503 // true

0o767 === 503 // true

**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.

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)]);

})

More info: [MDN Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

**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.

// No sample yet

More info: [MDN Reflect](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect)

**Tail Calls**

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

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 ES6

factorial(100000)