Hacé click acá para dejar tu feedback sobre esta clase.

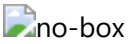
Hacé click acá completar el quiz teórico de esta lecture.

ECMAScript 6

El ES6 ,también conocido como ECMAScript 2015 o ES2015+, es la última versión del standart. Este nuevo update es el primero desde que se estandarizó el lenguaje en 2009. Las implementaciones del nuevo standart todavía se están haciendo para varios motores de JavaScript.

Historia

Desde su aparición en 1995, JS fue evolucionando lentamente. ECMAScript apareció como standart en 1997, y desde ahí viene lanzando nuevas versiones como ES3, ES5, ES6, etc..



Como ven entre ES3 y ES5 pasaron *10 años* y entre ES5 y ES6 pasaron *6 años*. Ahora la idea es lanzar nuevas versiones con cambios pequeños cada año.

> Podemos ver un *mapa* de las compatibilidades actuales de varios engines con respecto al nuevo standart acá

Como todavía no es compatible con muchos browsers, para poder utilizarlo vamos a utilizar una librería llamada `babel.js` que nos servirá para traducir código *ES6* a la versión actual para así mantener compatibilidad con los engines actuales.

Primero veamos algunas de las cosas que cambiaron en el nuevo standart.

Nuevas Features

ES6 incluye las siguientes features:

- let + const (Block Scoping)
- arrows =>
- classes
- object literals mejorados
- template strings
- destructuring
- default + rest + spread
- iterators + for..of
- generators
- unicode
- modules
- module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- promises
- math + number + string + array + object APIs
- binary and octal literals
- reflect api
- tail calls
- Optional Chaining

ECMAScript 6 Features

Let + Const

`let` es el nuevo `var`. Sólo que let tiene un scope distinto, este está declarado sólo dentro del `bloque` donde aparece y no dentro del scope (por ejemplo si uso `let` dentro de un `for` no voy a poder ver esa variable afuera del mismo, está bien claro acá). `const` es para declarar variables inmutables o sea que no pueden cambiar. Si queremos asignar un nuevo valor a una variable declara con `const` vamos a obtener un Error.

```
function f() {
  {
    let x;
    {
      // okay, block scoped name
      const x = "sneaky";
      // error, const
      x = "foo";
    }
    // error, already declared in block
    let x = "inner";
  }
}
```

Más Info: let statement, const statement

Arrows

Arrows (o flechas) son una forma de abreviar la declaración de una función y utiliza la sintaxis `=>` (está inspirada en sintaxis similares de C#, Java 8 y CoffeeScript). Estas soportan cuerpos que sean statements (ifs, fors, etc..) y también cuerpos que retornen el resultado de una expresion. A diferencia de las funciones normales, las funciones **arrows comparten el mismo `this` que el código que las rodea**.

```
// Cuerpos con Expresiones
var pares = impares.map(v => v + 1);
var nums = pares.map((v, i) => v + i);

// Cuerpos con Statements
nums.forEach(v => {
  if (v % 5 === 0)
    cincos.push(v);
});

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

Más información: MDN Arrow Functions

Classes

Las clases en ES6 son simplemente syntax sugar sobre el patrón de prototipado de objetos. Tener una forma particular de declarar clases, hace que sea más fácil de usar y mejora la interoperabilidad. Las clases soportan la herencia basada en prototipos, llamadas a super, instance y métodos estáticos y constructores.

```
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();
  }
}
```

Más info: MDN Classes

Object Literals Mejorados

Ahora los Object Literals se extendedieron para setear el prototipo durante su construcción, atajos para cuando hacemos una asignación de este tipo: `propiedad: propiedad`, definimos métodos, cuando hacemos llamadas a super, y al computar nombres de propiedades en una expresión. Todo esto junto mejora el proceso de diseño orientado a objetos ya que trabaja en sinergía con las mejoras en las clases antes mencionadas.

```
var obj = {
    // __proto__
    __proto__: theProtoObj, //extiende el prototipo
    propiedad, // atajo para propiedad:propiedad
    // Methods
    toString() {
     // Super calls
     return "d " + super.toString();
    },
    // Computed (dynamic) property names
    [ 'prop_' + (() => 42)() ]: 42
};
```

Más Info: MDN Grammar and types: Object literals

Template Strings

Los Template Strings son una syntax sugar para la construcción de strings. Es parecido a la interpolación de stings de Perl, Python y otros lenguajes.

```
// 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);
```

Más Info: MDN Template Strings

Destructuring

Las estructuras se pueden desestructurar para poder seleccionar valores usando patrones de matcheo, con soporte para arreglos y objetos. Esto funciona igual que buscar una propiedad de un objeto `foo['bar']` en el sentido que ambos son `fail-soft`, es decir, que producen un `undefined` cuando algo no se encuentra.

```
// 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;
```

Más Info: MDN Destructuring assignment

Default + Rest + Spread

Las declaraciones de funciones pueden tener argumentos que defaultean a un valor si no son declarados. También se puede transformar un arreglo en argumentos consecutivos. Y al reves, o sea podemos bindear los

últimos elementos de los argumentos como si fuera un sólo arreglo.

```
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
```

Más Info: Default parameters, Rest parameters, Spread Operator

Iterators + For..Of

Un objeto es un iterador cuando sabe como acceder a una colección de items de a uno, mientras mantiene su posición dentro de esa secuencia. En ES6 un iterador es cualquier cosa que provea un método `next()` que retorn el próximo item en la secuencia.

También se generalizó el `for...in` para poder usar `for...of` en iteradores.

```
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);
}
```

Más info: MDN for...of

Generators

Los iteradores son herramientas muy útiles, hay que tener mucho cuidado cuando los programamos porque necesitan mantener internamente su estado todo el tiempo. Los generadores nos proveen una alternativa poderosa: te dejan definir un algoritmo iterativo escribiendo una función que mantenga su propio estado.

```
function* idMaker(){
  var index = 0;
  while(true)
    yield index++;
}

var gen = idMaker();

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
// ...
```

Más info: MDN Iteration protocols

Unicode

Hicieron adiciones para mejorar el soporte con Unicode.

```
// 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);
}
```

Más Info: MDN RegExp.prototype.unicode

Modules

Ahora Javascript soporta módulos de forma nativa (antes era mediante `CommonJS`), es muy parecido ya que están basados en la filosofía de lo que ya veniamos usando.

```
// 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));
```

Ahora podemos distinguir los **named exports**, que son básicamentes todos los que exporten una variable con un nombre, por ejemplo: `export var pi = 3.14;`. Podemos tener muchos de estos por archivo. También existe el **Default Export**: Sólo puede haber *uno* por archivo, este es más parecido a la vieja forma de exportar. Generalmente lo usamos cuando, al importar, queremos que esté *todo* dentro de un objeto y no importar *varios* objetos separados.

```
// 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);
```

Más info: import statement, export statement

Map + Set + WeakMap + WeakSet

Nuevas Estructuras de datos. En general basadas en estructuras muy usadas en otros lenguajes.

```
// 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
```

Más Info: Map, Set, WeakMap, WeakSet

Subclassable Built-ins

En ES6, objetos nativos como `Array`, `Date` y elementos del `DOM` pueden ser heredados por una subclase.

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

Se agregaron cosas nuevas en las librerías nativas, incluyendo librerías matemáticas, funciones para convertir arreglos, funciones para trabajar Strings, y Object.assign para copiar objetos.

```
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) })
```

Más info: Number, Math, Array.from, Array.of, Array.prototype.copyWithin, Object.assign

Binary y Octal Literals

Dos nuevas formas literales agregadas para binario (b) y octal (o).

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

Promises

Promises es una librería para mejorar la programación asíncrónica. Las Promises son una representación de tipo first-class de un valor que va a estar disponible en el futuro. Esto también ya existia con otras librerías de terceros.

```
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)]);
})
```

Más info: MDN Promise

Tail Calls

Se puede implementar llamadas recursivas sin tener que agregar un un frame al `call stack` haciendo que sea segura la ejecución de una función recursiva (sin temer por el stack overflow).

```
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)
```

> Podemos ver una lista de features más detallada y comparada con la versión anterior aquí

Optional Chaining

El operador de encadenamiento opcional `?.` permite leer el valor de una propiedad ubicada dentro de una cadena de objetos conectados sin tener que validar expresamente que cada referencia en la cadena sea válida. El operador `?.` funciona de manera similar a el operador de encadenamiento `.` , excepto que en lugar de causar un error si una referencia es `null` o `undefined`, la expresión hace una short-circuit evaluation (`la evaluacion del segundo termino solo se efectua si el primero no permite definir un resultado ej: (true ||false)---> da verdadero sin siquiera pasar por false`) con un valor de retorno de `undefined`. Cuando se usa con llamadas a funciones, devuelve `undefined` si la función dada no existe. Esto da como resultado expresiones más cortas y simples cuando se accede a propiedades

encadenadas dónde existe la posibilidad de que falte una referencia. También puede ser útil al explorar el contenido de un objeto cuando no hay una garantía conocida de qué propiedades se requieren.

```
const adventurer = {
  name: 'Alice',
  cat: {
    name: 'Dinah'
  }
};

const dogName = adventurer.dog?.name;
console.log(dogName);
// expected output: undefined

console.log(adventurer.someNonExistentMethod?.());
// expected output: undefined
```

Compatibilidad

Para poder utilizar la sintaxis y las nuevas funcionalidad de ES6 en los motores no compatibles, vamos a utilizar Babel.js

Para instalarlo:

```
# install the cli and this preset
npm install --save-dev @babel/core @babel/cli

# make a .babelrc (config file) with the preset
npm install @babel/preset-env
echo '{ "presets": ["@babel/preset-env"] }' > .babelrc

# create a file to run on
echo 'console.log([1, 2, 3].map(n => n + 1));' > index.js

# run it
./node_modules/.bin/babel-node index.js
```

De esa forma vamos a poder transformar un archivo y lo tenemos que hacer manualmente. Si queremos que sea automático tenemos muchísimas opciones. Veamos algunas en la página de babel y elijamos el setup que más nos guste.

Homework

Completa la tarea descrita en el archivo README