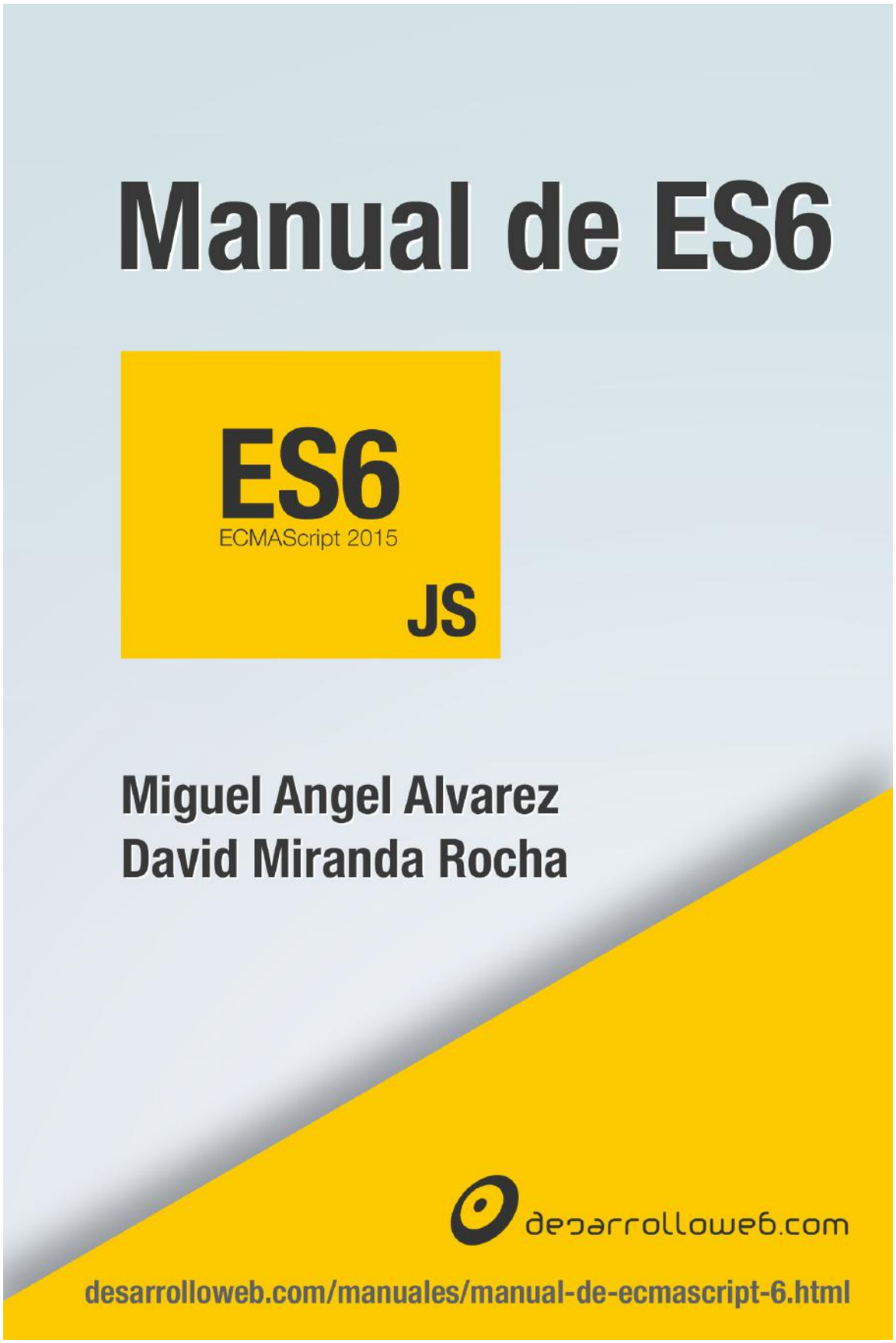

Manual de ES6
ES6
ECMAScript 2015
JS
Miguel Angel Alvarez
David Miranda Rocha
desarrolloweb.com
desarrolloweb.com/manuales/manual-de-ecmascript-6.html

Introducción: Manual de ES6

En este manual vamos a abordar el aprendizaje de ECMAScript 2015, la versión 6 del estándar de Javascript conocida como ES6, que nos ofrece nuevas construcciones, alternativas de organizar el código y herramientas para hacer de Javascript un lenguaje más productivo y sencillo de usar.

En este manual queremos abordar tanto las novedades del lenguaje, en cuanto a sintaxis, estructuras de control, etc., así como la manera de usar ES6 ya mismo valiéndonos de los transpiladores.

Iremos publicando nuevos artículos en este manual durante las próximas semanas para ofrecer una guía completa a los programadores que busquen mejorar su experiencia de desarrollo con Javascript. De momento puedes ver los primeros publicados, para ir abordando el tema, junto con referencias para el aprendizaje que encontrarás en los artículos y que te servirán para complementar el material.

Encuentras este manual online en:
http://desarrolloweb.com/manuales/manual-de-ecmascript-6.html

Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.

David Miranda Rocha

Desarrollador web especializado en Javascript, sus mejores herramientas AngularJS y Polymer.

Introducción a ES6

Qué es ES6 y cómo podemos estudiarlo. En esta introducción veremos qué papel juega el estándar ECMAScript y cómo es su versión de 2015, junto con nuestros consejos para estudiarlo. Esta guía está pensada para personas que ya conozcan Javascript.

Qué es ES6 y cómo estudiarlo

Qué es ES6, también conocido como ECMAScript 2015, el más reciente estándar del popular lenguaje de programación Javascript.

En este artículo pretendemos dejar las cosas claras en cuanto a conceptos que el estudiante debe conocer sobre ECMAScript y su reciente versión de 2015, llamada habitualmente por su abreviación ES6. Conoceremos qué es ECMAScript, qué es ES6 y bajo qué contextos podemos usar esta versión del lenguaje.

El objetivo es que después de la lectura de este artículo cualquier desarrollador sepa cómo puede comenzar a usar las últimas mejoras de Javascript y dónde encontrar mayores informaciones para documentarse o crear su entorno de trabajo.

Comenzaremos por un rápido análisis de los conceptos principales que pueden ser nuevos para el lector.

Qué es ECMAScript

ECMAScript es el estándar que define cómo debe de ser el lenguaje Javascript. Sin entrar en datos históricos, que puedes consultar otras fuentes como la Wikipedia sobre ECMAScript, cabe decir que es la especificación que los fabricantes deben seguir al crear intérpretes para Javascript.

Como sabes, Javascript es interpretado y procesado por multitud de plataformas, entre las que se encuentran los navegadores web, así como NodeJS y otros ámbitos más específicos como el desarrollo de aplicaciones para Windows y otros sistemas operativos. Todos los responsables de cada una de esas tecnologías se encargan de interpretar el lenguaje tal como dice el estándar ECMAScript.

Cómo son las versiones del estándar de Javascript

A lo largo de la existencia de Javascript y desde que se le encargó a ECMA el trabajo de su estandarización, han aparecido diversas versiones de Javascript con cada vez mayores funcionalidades, estructuras de control, etc. Esas son las versiones del estándar y hoy de la que nos ocupamos es de ES6, la sexta versión.

ES5 estuvo con nosotros durante muchos años y a día de hoy es la versión de Javascript más extendida en todo tipo de plataformas. Cuando alguien dice que conoce o usa Javascript es común entender que lo que usa es ES5, el Javascript con mayor índice de compatibilidad. De ésto también se entiende que, cuando queremos escribir un código compatible con todos los navegadores o sistemas, lo normal es que ese código sea ES5.

Sin embargo, hoy muchos sistemas son capaces de entender ES6, la versión del estándar ECMAScript presentada en 2015. ES6 mejora varios aspectos del lenguaje y proporciona nuevas variantes a la programación, que nos aportan distintas ventajas.

El mundo de las versiones no se queda ahí y hoy ya existen borradores de lo que será ES7, que todavía nos traerá nuevas construcciones en el lenguaje, haciéndolo más productivo y sofisticado.

Transpiladores

Conscientes de los problemas de compatibilidad o soporte a ES6 en las distintas plataformas, un desarrollador poco informado podría pensar que:

1. Sea poco aconsejable usar hoy ES6, debido a la falta de compatibilidad.
2. Lo correcto sería esperar a que todos los navegadores se pongan al día para empezar a usar ES6 con todas las garantías.

Afortunadamente, ninguna de esas suposiciones se ajusta a la realidad. Primero porque si ES6 nos aporta diversas ventajas, lo aconsejable es usarlo ya. Luego porque es absurdo quedarse esperando a que todos los navegadores soporten Javascript en la versión ES6. Quizás nunca llegue ese momento de total compatibilidad o posiblemente para entonces hayan sacado nuevas versiones del lenguaje que también deberías usar.

Así que, para facilitar nuestra vida y poder comenzar a usar ES6 en cualquier proyecto, han surgido los transpiladores, una herramienta que ha venido a nuestro kit de desarrollo para quedarse.

Los transpiladores son programas capaces de traducir el código de un lenguaje para otro, o de una versión para otra. Por ejemplo, el código escrito en ES6, traducirlo a ES5. Dicho de otra manera, el código con posibles problemas de compatibilidad, hacerlo compatible con cualquier plataforma.

El transpilador es una herramienta que se usa durante la fase de desarrollo. En esa fase el programador escribe el código y el transpilador lo convierte en un proceso de "tradución/compilación = transpilación". El código transpilado, compatible, es el que realmente se distribuye o se despliega para llevar a producción. Por tanto, todo el trabajo de traducción del código se queda solo en la etapa de desarrollo y no supone una carga mayor para el sistema donde se va a ejecutar de cara al público.

Hoy tenemos transpiladores para traducir ES6 a ES5, pero también los hay para traducir de otros lenguajes a Javascript. Quizás hayas oído hablar de TypeScript, o de CoffeeScript o Flow. Son lenguajes que una vez transpilados se convierten en Javascript ES5, compatible con cualquier plataforma.

Cómo abordar ES6

Esperamos haberte convencido de la conveniencia de usar ES6. Si es así te preguntarás cómo comenzar a aprender las nuevas características del lenguaje y cómo usar esos transpiladores de una manera ágil para que se encarguen de la parte de la traducción, sin que ello merme tu productividad o dificulte la operativa del desarrollo.

Tanto aprender ES6 como crear tu entorno de trabajo es algo que ya hemos introducido en otros materiales publicados en DesarrolloWeb.com, que ahora te resumimos.

Manual de ES6: Este artículo será el primero de un manual en el que vamos a explicar todo lo que necesitas saber sobre ES6. En este manual vamos a explicar primero las novedades del lenguaje y luego nos detendremos a crear nuestro entorno de trabajo.

Workflow para trabajo con Babel: En este artículo te ofrecemos los primeros pasos para configurar Babel, uno de los transpiladores disponibles, el más usado en la actualidad.

Vídeos de ECMAScript 6: Tenemos varios vídeos publicados sobre ES6 y el uso de transpiladores. Los puedes encontrar en la lista de ECMAScript 6.

Curso completo de ES6: Si quieres aprender de manera completa, cómoda y rápidamente, también deberías conocer el Curso Completo de ES6 que encontrarás EscuelaIT. Este curso aborda todo lo que necesitas saber sobre el estándar y sobre las herramientas que te permitirán usarlo ya mismo con el mejor flujo de desarrollo posible.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *16/08/2016*
Disponible online en *http://desarrolloweb.com/articulos/que-es-ecmascript6-como-estudiarlo.html*

Probar y experimentar ES6 en distintas plataformas: compatibilidad

Cómo puedes probar tus ejemplos de código realizados con ES6, de modo que puedas aprender las novedades de Javascript sin tener que preocuparte con el uso de transpiladores.

Apenas empezamos el Manual de ES6 y antes de comenzar a ver código queremos explicar nuestro planteamiento para que cualquier persona pueda iniciarse en el aprendizaje. En concreto queremos explicar brevemente cuáles son las alternativas que tenemos disponibles para poder probar ejemplos realizados en ECMAScript 2015 (ES6) sin tener que complicarnos la vida creando un entorno de trabajo que use transpiladores.

Los transpiladores en si no son difíciles de usar, pero sí que requieren un tiempo de configuración para la puesta en marcha y estamos seguros que, como nosotros, estaréis ansiosos en poner código en funcionamiento y realizar nuestras primeras pruebas.

Así que, en nuestro Manual de ES6 vamos a seguir este esquema:

- Primero vamos a comenzar explicando las novedades del lenguaje
- Luego veremos cómo configurar un entorno de desarrollo usando transpiladores

Cómo probar ya mismo Javascript ES6

A continuación puedes encontrar las alternativas para ejecutar código Javascript ES6 sin usar transpiladores. Básicamente se dividen entre la posibilidad de usar la ejecución en el entorno de un navegador o en el entorno de propósito general NodeJS.

Navegadores compatibles con ES6: En cuanto a navegadores, el navegador que incorpora un porcentaje mayor de las características de ES6 en estos momentos es Safari 10, con un 99%. Como no todo el mundo tiene un Mac, plataforma donde corre Safari, la alternativa más fiable sería Google Chrome, con un 97% de las nuevas funciones. El nuevo navegador de Microsoft Edge, sería la siguiente alternativa más compatible con un 95% de soporte. Por su parte, Firefox tiene implementadas un 92% de las novedades del lenguaje.

> **Nota:** Este dato es en el momento de escribir este artículo, correspondiente a las últimas versiones de navegadores, aunque en el futuro, lógicamente, se irá obteniendo más soporte.

En resumen usa Google Chrome para probar ES6 y tendrás casi el soporte total. Sin embargo, aunque estos números sean altos, no podemos asegurar que todos los ordenadores de los usuarios tienen actualizados los navegadores, por eso para entornos de producción siguen siendo indispensables los transpiladores que veremos también más tarde en este manual.

NodeJS: La otra alternativa de probar tus ejemplos con ES6 es ejecutar los ejemplos en la plataforma NodeJS. Ya sabes, "Node" es el Javascript sacado fuera del navegador, que permite realizar todo tipo de programas o aplicaciones de propósito general.

La penetración de ES6 en NodeJS depende de la versión. A partir de la 0.12 o superior ya se va encontrando soporte (21%) y éste crece a medida que crece la versión. La mejor alternativa es instalar la versión estable más reciente, que sería en estos momentos la 6.x, que tiene soporte del 92% de las novedades del estándar.

> **Nota:** Puedes saber qué versión de Node tienes instalada con el comando de consola "node -v". Si no sabes instalar NodeJS o no sabes crear y ejecutar tus programas Javascript mediante la plataforma NodeJS, te recomendamos leer los primeros capítulos del Manual de NodeJS.

Tabla de compatibilidad ES6 con las distintas plataformas

Existen muchas más plataformas para la ejecución de Javascript, tanto en lo que respecta a navegadores e incluso la ejecución de Javascript del lado del servidor o en programas de propósito general al estilo de NodeJS. Si quieres verificar su soporte a ECMAScript 2015 en la actualidad puedes consultar la tabla de compatibilidad en esta URL: http://kangax.github.io/compat-table/es6/

Está muy completa y detallada no solo en lo que respecta a las distintas plataformas o sistemas operativos, como a cada uno de los items que ES6 agrega como estándar a Javascript. También encontrarás el nivel de compatibilidad que adquieres usando pollyfills o transpiladores como Babel.

Conclusión

Probar las novedades de Javascript incorporadas por ES6 está al alcance de todos, ya que la mayoría están implementadas en navegadores de uso común.

Ya que es así, vamos a comenzar nuestro Manual de ES6 explicando estas novedades del lenguaje, comenzando por las de mayor importancia. En los siguientes artículos iremos más a la práctica.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *29/08/2016*
Disponible online en *http://desarrolloweb.com/articulos/probar-experimentar-es6-compatibilidad.html*

Primeros pasos con ECMAScript 2015 (aka ES6)

Qué es ES6, o ECMAScript 2015, el estándar que define el lenguaje Javascript. Primeros pasos para usar ES6 en cualquier sitio web manteniendo la compatibilidad con todos los navegadores del mercado.

Como profesional en la web debes saber lo importante que es conocer las tecnologías de desarrollo y sus novedades, especialmente aquellas que son estándares y que por tanto cualquier proyecto utilizará. En el caso de Javascript las novedades vienen marcadas por el estándar ECMAScript, que recientemente ha presentado su sexta versión, ES6, también conocido como ECMAScript 2015.

Con este texto comenzamos una serie de artículos donde iremos analizando con detalle qué nos aporta ES6 y cómo implementarlo en nuestros proyectos. De momento en esta entrega explicaremos con detalle en qué consiste este estándar y cómo integrarlo en un proyecto para la web, de modo que aseguremos que nuestro código sea compatible con todos los navegadores que lo puedan ejecutar.

Javascript Vs ECMAScript

Cuando nos acercamos a ECMAScript a muchos de nosotros nos surge la duda sobre qué es y qué diferencia tiene con Javascript. Así que comenzaremos rápidamente aclarando este punto.

Javascript es una marca comercial propiedad de Oracle y antes de Sun Microsystems. Cuando los ingenieros de Netscape incorporaron un lenguaje de Scripting en su navegador decidieron, después de varios nombres, denominarlo Javascript bajo consentimiento de Sun.

Más tarde Netscape decidió proponer su lenguaje como estándar y el trabajo fue realizado por ECMA, dando lugar a ECMAScript. Por tanto, hoy Javascript sería un dialecto de ECMAScript.

A lo largo de los años se han ido incorporando novedades al estándar, que más tarde o temprano son aplicadas a los navegadores en sus propias implementaciones. Hacía mucho que no se publicaba ninguna novedad o mejora, exactamente desde el 2009 donde se hacía pública su edición número 5, que en los últimos años se ha convertido en un gran referente.

Hoy Javascript, y por tanto ECMAScript, se encuentra implementado en navegadores y más recientemente como lenguaje de propósito general como NodeJS, en el que no vamos a entrar en detalle, ya que tenemos un excelente curso y diversos artículos en el manual de Node donde nos adentramos más en el.

Compatibilidad de ECMASCRIPT 2015

Si queremos usar la versión más actual de ECMAScript, la primera pregunta que uno se suele formular es la compatibilidad con las distintas plataformas donde está disponible.

Nota: Como hemos dicho, encontramos Javascript tanto en navegadores como en lenguaje de propósito general, o de servidor, mediante la plataforma NodeJS. La respuesta entonces es particular para cada caso.

- NodeJS: Podemos usar ES6 en NodeJS y muchas de las funcionalidades están disponibles de forma nativa. Más adelante hablaremos en detalle de Babel, pero ya te adelantamos que existe una distribución reducida de Babel para poder trabajar con NodeJS.
- Navegadores: No ocurre igual con los navegadores, ya que la implementación de Javascript en cada uno de ellos depende del fabricante del navegador.

Apartaremos entonces la discusión de compatibilidad en NodeJS. En las próximas líneas estamos hablando siempre compatibilidad en navegadores. Además en el vídeo que hay al final de este texto también se hace referencia a otras utilidades para trabajar con ES6 en NodeJS.

En los navegadores se está extendiendo progresivamente el soporte a ES6, pero aún queda trabajo por hacer. A la larga todos los navegadores actualizados implementarán el estándar, pero el momento en que lo hagan depende de los fabricantes. Los hay que llevan más retraso también encontramos fabricantes que incluso que se están adelantando y han comenzado a dar soporte a la siguiente versión, ES7.

Esta situación es perfectamente normal, por lo que no debe decepcionarnos. Ni tampoco es motivo por el cual no se pueda usar ES6 ya mismo, ya que existen diversas herramientas que nos solucionarán la vida y nos permitirán usar todas las posibilidades de ECMAScript 6 en la actualidad.

Transpiladores

La palabra "transpilador" no la encontrarás en el diccionario. Se ha acuñado recientemente a raíz de las necesidades de traducción diversas versiones de ECMAScript y dialectos y lenguajes derivados. Transpilador es realidad es una adaptación al español del término "Transpiler", que a su vez viene de "Tanslator" y "Compiler".

Los "transpiladores" son por tanto una especie de compiladores que transforman nuestro código ES6 a ES5, para así ser totalmente compatible con todos los navegadores. Uno de los más conocidos y usados es Babel, pero también tenemos otros como Google Traceur.

En general y por decirlo con palabras sencillas, no son más que traductores del "Javascript del mañana" al "Javascript de hoy". Todos tienen la particularidad que funcionar en la etapa de desarrollo. Es decir, la traducción de tu código ES6 a ES5 se realiza en tu ordenador, generando automáticamente código ES5 a partir de tu código ES6. Esa traducción se realiza antes de desplegar un proyecto, de modo que, cuando subes tus archivos JS al servidor, tienen únicamente código Javascript que es entendible por todos los navegadores. Por tanto, no tienes por qué preocuparte de rendimiento, puesto que es una tarea que se realiza una vez, y no cada vez que se ejecuta un script, como ocurre con el compilado de los lenguajes no interpretados.

El código generado contiene un mapeado al código que tú has escrito, de modo que todo el trabajo de depuración del programa se realiza sobre el código ES6 y no sobre el código generado. Así pues, todo el flujo de desarrollo y pruebas se realiza contra tu propio código, sin importarte cómo se haya traducido a ES5. Dicho de otro modo, esta traducción es realmente transparente para el desarrollador, tú seguirás desarrollando como si todo navegador entendiese ES6.

Un transpilador puede transformar nuestro código ES6:

```
const nomVariable = n => n * n;
```

A este otro código ES5:

```
var nomVariable = function nomVariable (n) {
  return n * n;
};
```

Podéis apreciar el cambio entre versiones del estándar y el ahorro de líneas y la mayor facilidad de organizar nuestro código con esta nueva sintaxis de ES6.

Hay muchas novedades en las que nos iremos adentrando, pero en este momento nos hace falta crear un entorno de trabajo en el que podamos desarrollar en ES6 sin problemas, configurando nuestro propio transpilador. Lo veremos paso a paso a continuación.

Crear entorno de trabajo para ES6 con Babel

Ahora vamos a comentar varias maneras de iniciar y compilar con este magnífico transpilador conocido por

Babel.

> **Nota:** Existen practicamente integraciones con casi todas las librerías/frameworks populares, puedes verlo mas detallado en http://babeljs.io/docs/setup/.

Para poder trabajar con Babel necesitamos instalarlo en nuestro equipo, para ello necesitamos tener Node y npm si no sabes de qué hablamos o simplemente no lo tienes instalado puedes ver nuestra guía en http://www.desarrolloweb.com/manuales/manual-nodejs.html.

Babel-cli (Opción de instalación global)

Instalamos las herramientas de trabajo con Babel mediante línea de comandos de forma global mediante nuestra consola:

```
npm install --global babel-cli
```

Babel viene construido modularmente y en este caso el que nos interesa es el modulo de ES6, así que procedemos a instalar ya en nuestro directorio raíz nuestro preset:

```
npm i -D babel-preset-es2015
```

Ya solo nos queda indicárselo a babel creando el fichero .babelrc con el siguiente contenido:

```
{
  "presets": ["es2015"]
}
```

Después de esto ya podemos compilar nuestro fichero .js con el comando

```
babel mi-fichero.js
```

Esto compilará nuestro js, mostrando el resultado directamente en la pantalla del terminal. Si estamos contentos con el resultado o simplemente queremos ahorrarnos este paso y compilar directamente en un fichero debemos escribir en el terminal

```
babel mi-fichero.js -o mi-fichero-compilado.js
```

Si por el contrario tenemos varios ficheros js que queremos llevar a producción podemos compilar todos los ficheros de un directorio con las instrucciones

```
babel la-carpeta -d lib
```

```
Babel-cli (Opción de instalación local)
```

Si tenemos varios proyectos que necesitan versiones de babel diferentes podemos instalarlo de forma local. Es algo interesante, pero realmente no todos los desarrolladores lo prefieren, por la comodidad de instalarlo una única vez. Suele ser más recomendable la instalación local, ya que independiza las dependencias por proyectos, pudiendo usar diferentes versiones en cada uno, ya sea de Babel o con cualquier otra dependencia que necesitemos. En este caso primero debemos ir a la carpeta raíz del proyecto, con nuestra consola, e introducir el comando:

```
npm install --save-dev babel-cli
```

Si en un momento dado deseamos desinstalar nuestra copia global para usar nuestra copia local ingresamos en la consola

```
npm uninstall --global babel-cli
```

Una vez que acabe de instalar nos quedaría un package.json parecido a este

```
{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "devDependencies": {
    "babel-cli": "^6.0.0"
  }
}
```

Ahora, en vez de ejecutar Babel directamente desde la línea de comandos, podemos escribir nuestros comandos en **npm scripts** los cuales usarán nuestra versión local. Simplemente se añade el campo "scripts" a el mencionado paquete package.json situado en la raíz del proyecto. En dicho campo se debe poner el comando de babel, como build.

Procedemos a instalar el preset para ES6:

```
npm i -D babel-preset-es2015
```

Introducimos su configuración en nuestro package.json quedando así:

```
{
    "name": "mi-proyecto",
    "version": "1.0.0",
```

```
    "scripts": {
      "build": "babel directorio -d lib"
   },
"babel":{
  "presets": ["es2015"]
},
    "devDependencies": {
      "babel-cli": "^6.0.0"
    }
  }
```

Ahora, desde nuestro terminal, podemos correr lo siguiente:

```
npm run build
```

Esto ejecutará Babel igual que antes, sólo que ahora estamos usando una copia local.

> **Nota:** Toda esta información está sacada de su documentación oficial que podéis consultar en GitHub Babel Handbook.

Como podéis comprobar, resulta bastante fácil dar el primer paso hacia un código más limpio, estructurado y mucho más fácil de crear y leer. En los siguientes artículos entraremos más profundamente en las novedades de ECMAScript 2015. Todo esto es material de estudio también del curso de ES6 de EscuelaIT, que está a punto de empezar.

Recordar que Babel también está integrado con automatizadores de tareas como Gulp que nos facilitan mucho la operativa del trabajo durante el desarrollo de un proyecto.

En resumen, no estamos hablando de una librería o un framework que pueda estar de moda, sino de las bases de un lenguaje tan usado como Javascript, con infinitas salidas profesionales. Como desarrolladores es nuestra responsabilidad usar las tecnologías de la mejor manera y si estamos buscando nuevas posibilidades en el mercado laboral es algo que debemos conocer para estar por delante de otros candidatos.

> Para ver este vídeo es necesario visitar el artículo original en:
> http://desarrolloweb.com/articulos/primeros-pasos-ecmascript-2015.html
>
> Este artículo es obra de *David Miranda Rocha*
> Fue publicado por primera vez en *25/02/2016*
> Disponible online en *http://desarrolloweb.com/articulos/primeros-pasos-ecmascript-2015.html*

Novedades a Javascript introducidas por ES6

En los siguientes capítulos iremos repasando las novedades que aporta ES6. Nos iremos centrando en todas aquellas utilidades que resultan más interesantes para los desarrolladores y por tanto, que son más utilizadas en el día a día con Javascript.

Let y const: variables en ECMAScript 2015 (ES6)

Descubramos las nuevas maneras de declarar variables en Javascript, let y const, sus ámbitos y comportamientos definidos por el estándar ECMAScript 2015 (ES6).

En este artículo vamos a comenzar a abordar las características de ES6 en cuanto a novedades que aportan en el lenguaje Javascript. De una manera práctica iremos abordando las nuevas posibilidades del lenguaje en el estándar definido en 2015. En este caso veremos dos nuevas formas de tratar las variables, let y const.

Recuerda que si quieres saber más de dónde viene ES6, cómo usarlo y qué implica, debes consultar el Manual de ES6. Pero antes de meternos de lleno en las cosas que nos aporta ES6 debemos recordar que en Javascript solo teníamos una forma de declarar variables. Se hacía con la palabra clave "var" seguida del nombre de variable a declarar (o variables separadas por comas). Sería algo como esto:

```
var variable;
```

Esta variable podía ser de ámbito global o local. Su ámbito depende del lugar donde se declara pero las posiblidades eran bien limitadas. Para hacer una variable global la teníamos que declarar simplemente fuera del cuerpo de una función. En cambio, las variables locales eran declaradas dentro de una función y solo eran accesibles dentro del código de esa función.

En resumen, en ES5 (Javascript más tradicional) no se permitía declarar un alcance más acotado, como las llaves de una condición if, un bucle etc. Veamos un ejemplo:

```
function miFuncion() {
  console.log(miVar);
```

```
  if (true) {
    var miVar = "Hola mundo";
  }
  console.log(miVar);
};
```

En este caso, pensando en ES5, la variable "miVar" está definida dentro de un bloque if, pero aun así su ámbito será el de la función miFuncion(). Si lo ponemos en ejecución el resultado que obtendremos en consola es el siguiente:

```
undefined
Hola mundo
```

Este resultado se debe a que en la primera llamada miVar ya existe pero no tiene valor definido, y en la segunda ya tiene un valor definido “Hola mundo”.

Quizás lo que sería de esperar es que la variable miVar quedase limitada al ámbito donde fue declarada, dentro de las llaves del if. Pero lo cierto es que se extiende al ámbito de toda la función.

El código anterior, en ES5, equivaldría al siguiente:

```
function () {
var miVar;
  console.log(miVar);
  if (true) {
    miVar = "Hola mundo";
  }
  console.log(miVar);
};
```

Como podemos observar, en cuanto ve una variable ya sea parte de un if o de la propia función pasa a declararla antes de seguir ejecutando el código (Esto es debido a la doble pasada que hace por el código el intérprete de Javascript). Por eso, en el primer console.log nos sale como declarada pero sin valor definido. Este tipo de cosas son caballo de batalla a diario con Javascript y seguro que a más de uno le han provocado unos cuantos dolores de cabeza.

Declaración de variables con Let

Para solucionar todo esto en Javascript ES6 se ha implementado alguna forma nueva de declarar variables que ahora será la más indicada en la mayoría de los casos. Se trata de la construcción "let" que veremos mejor con un ejemplo similar:

```
function () {
  console.log(miVar);
  if (true) {
    let miVar = "Hola mundo";
  }
```

```
  console.log(miVar);
};
```

La clave de let es que restringe su existencia al ámbito donde ha sido declarada (cualquier ámbito expresado con unas llaves). Por tanto, la salida de la consola sería en este caso:

```
undefined
undefined
```

Por qué este cambio? Como hemos dicho, let es la nueva forma de declarar variables de bloque, sólo existirán dentro del bloque que las contenga y no contaminará nunca nuestro código a su alrededor. Es una gran aportación para no tener que estar pendiente de sobreescribir variables y con lo que al final te juntabas con un buen número de ellas, este ejemplo es de un if, pero es lo mismo con cualquier bucle, sea un while, for, etc.

Declaración const

Otra nueva forma de crear variables es "const". Ésto nos crea una variable donde su valor es inalterable a lo largo del código. Por ejemplo en un script para contabilidad sería una buena idea poner por ejemplo el IVA cuyo valor no se alterará en todo el programa ya que siempre debería ser el mismo. Otro ejemplo sería el valor "PI". Ya se sabe que en Javascript lo obtenemos mediante Math.PI, pero si lo tuviéramos que declarar lo ideal sería hacerlo como constante, ya que nunca cambia su valor.

> **Nota:** Por convección se acostumbra a definir en mayúscula los nombres de las constantes, para una mejor identificación, pero no es más que eso, una convención, pasemos a un ejemplo:

```
const MIVARIABLE = 1;
console. log(MIVARIABLE);
```

```
MIVARIABLE = 10;
console.log(MIVARIABLE);
```

El primero nos dará 1 que es su valor, pero el segundo nos dará un error y nos dirá que su valor sigue siendo 1. Obviamente, el lenguaje no nos permite modificar el valor de una constante.

La declaración const tiene ciertas unas peculiaridades interesantes. Por ejemplo:

```
const MIVARIABLE;
MIVARIABLE = 10;
console.log(MIVARIABLE);
```

En este caso hemos declarado MIVARIABLE y en la siguiente línea le hemos asignado un valor. Quizás parece correcto, pero cuando intentamos ponerle el valor, de manera posterior a su definición, vemos que no es posible por ser una constante. En realidad la constante se ha creado con el valor undefined y se quedará con ese valor por siempre. Es algo a tener muy en cuenta y que puede llevar a error muy fácilmente.

Declaración const en objetos

Otra particularidad de const es en su uso con objetos. Como imaginamos creará un objeto que su valor será inalterable y es así pero quizás no exactamente de la manera que pensamos. Veamos un ejemplo que lo dejará muy claro:

```
const OBJ = {
  id: '01',
  name: 'David'
}
OBJ.name = 'Miguel';
console.log(OBJ.name);
```

Esto nos imprimirá Miguel ya que la constante es el propio objeto no sus propiedades, fijaros que el objeto en sí está con mayúsculas pero sus propiedades no, porque no son constantes. Las propiedades del objeto en si se pueden modificar, añadir o borrar según necesitemos, lo que nunca podremos cambiar es la referencia a ese objeto. Por tanto, no podremos hacer es algo como: OBJ = 'Hola mundo'; ya que si estamos intentando alterar la referencia al objeto en sí.

Como podéis ver se abren muchas puertas a un código más sencillo y más práctico del que estábamos acostumbrados todo depende de nosotros el hacer un buen uso de él.

¿Para qué usar la antigua declaración "var" a partir de ahora?

Como habrás deducido, la mayoría de las ocasiones se usará "let" para declarar variables, dado que su ámbito es más restringido (al bloque definido por las llaves) y por tanto nos puede dar lugar a menos errores debidos a la interferencia entre variables del mismo nombre. Sin embargo, la declaración "var" sigue existiendo.

Obviamente, "var" se podrá usar cuando se trata de asignar un ámbito de toda una función, o ámbito global. Al principio del artículo hemos hablado de ámbitos de var, así que debe haber quedado más o menos claro.

Una posible regla sería: usar "let" siempre por regla general, hasta que por su ámbito restringido no nos venga bien, y entonces pasaremos a usar "var".

Otra cosa interesante que nos sirve para definir las diferencias entre let y var es el mecanismo de doble pasada del intérprete de Javascript. En la primera pasada leerá todas las variables definidas con "var" y en la segunda al ejecutar el código ya conocerá esas variables y las podrá usar. Así pues, una declaración "var" nos permite usar una variable que ha sido declarada más adelante en el código, siempre que su ámbito lo permita.

Este artículo es obra de *David Miranda Rocha*
Fue publicado por primera vez en *06/09/2016*
Disponible online en *http://desarrolloweb.com/articulos/conociendo-variables-ecmascript.html*

Literales de objeto en ES6

Interesantes mejoras en la sintaxis de creación de literales de objetos de Javascript introducidas por ES6.

En este artículo vamos a continuar nuestro manual de ES6 explicando algunas de las mejoras que ECMAScript 2015 (ES6) ha introducido en el lenguaje Javascript. En este caso le toca el turno a los literales de objeto.

Veremos varias mejoras, casi todas "azucar sintático", a la hora de escribir objetos, sus propiedades, métodos, valores, etc. Es una lección fácil de aprender pero que puede mejorar un tanto la agilidad con la que escribes código, a la vez de introducir nuevas posibilidades que antes no existían.

Supongo que a estas alturas no hay necesidad de mencionarlo, pero por si alguien se ha perdido, un literal de objeto no es más que un objeto escrito directamente en el código. Algo muy típico en la programación Javascript, que nos permite crear objetos indicado su valor y sin la necesidad de utilizar clases como "molde".

```
var literalObjeto = {
  campo: "valor",
  meGustaDesarrolloWeb: true,
  matriz: [ 1, 2, 5 ],
  metodo: function() {
    console.log('Esto es un método')
  }
}
```

Puedes encontrar más información sobre esto en el artículo literales de objeto en Javascript.

Nueva manera de escribir métodos

Una de las mejoras consiste en la posibilidad de resumir la escritura de métodos de los objetos literales. Con esta nueva sintaxis nos ahorramos la palabra "function", quedando de esta manera.

```
let factura = {
  id: 'A2016-156',
  cliente: 'Nombre del cliente',
  productos: [
    {
      name: 'Bombillas LED',
      precio: 234
    },
    {
      name: 'Tornillos inox',
      precio: 4
    }
  ],
  precio() {
    console.log('calculando precio');
    var precio = 0;
    this.productos.forEach(function(prod){
       precio += prod.precio
    })
    return precio
  }
}
```

Nota: Como puedes observar, dentro de un método podemos acceder a "this", que contiene una referencia al objeto actual al que pertenece este método.

Abreviación de propiedades

Más azúcar sintáctico para la definición de propiedades que responden a un patrón determinado, en el que el nombre de la propiedad es el mismo que la variable donde se guarda su valor. Es más fácil de ver que de explicar. Imagina que tienes ciertas variables:

```
var nombre = 'Pepe';
var edad = 35;
var saludar = function() {
  alert('hola')
}
```

Ahora podrías crear un objeto persona usando esas variables como sus propiedades.

```
var objPersona = {
  nombre: nombre,
  edad: edad,
  saludar: saludar
}
```

Este código es correcto en ES5 y nos dará como resultado un objeto como este:

```
{
  edad: 35,
  nombre: "Pepe",
  saludar: function () {
    alert('hola')
  }
}
```

Hasta ahí nada nuevo. Sin embargo, ahora en ES6 podemos resumir esa declaración del objeto, ya que el nombre de la propiedad corresponde con el nombre de la variable que contiene el valor a asignarle.

La nueva sintaxis para la abreviación de propiedades quedaría de esta manera:

```
var objPersona = {
  nombre,
  edad,
  saludar
}
```

Al ver el código, quien está acostumbrado a ES5, seguramente le parecerá que está faltando algo, pero es justamente esa la mejora, con menos código conseguimos lo mismo.

> **Nota:** Este patrón de construcción de objetos, en el que las propiedades toman valores de variables con el mismo nombre, es bastante habitual. Sobra decir que el objeto resultante es exactamente el mismo.

Propiedades con nombres computados

Esto es una nueva característica del lenguaje, aunque no deja de ser algo que podíamos hacer anteriormente de una manera un poco más rebuscada.

Se trata de crear un literal de objeto donde el nombre de una de sus propiedades es conmutado, es decir, una o varias de sus propiedades tienen nombres que son definidos en función del contenido de variables, en tiempo de ejecución.

El siguiente ejemplo te muestra lo que queremos decir. A través de las variables construimos un objeto, donde los nombres de las propiedades se generan a partir de los valores de esas variables.

```
var elemento = 'agua'
var modalidad = 'mariposa'
function deporte(){
   return 'natacion'
}
```

```
var registroDeporte = {
  [deporte()]: elemento,
  [modalidad + '100m']: 'Esta es la modalidad de 100 metros mariposa',
  [modalidad + '200m']: 'Esta es la modalidad de 200 metros mariposa',
}
```

Esto produciría un objeto como el siguiente:

```
{
  mariposa100m: "Esta es la modalidad de 100 metros mariposa",
  mariposa200m: "Esta es la modalidad de 200 metros mariposa",
  natacion: "agua"
}
```

En ES5 podríamos conseguir lo mismo, pero la sintaxis queda un poco más extraña, ya que necesitamos crear las propiedades accediendo a ellas como si fuera un array. Para ello primero debemos crear un objeto vacío y a continuación generamos sus propiedades.

```
var registroDeporte = {}
registroDeporte[deporte()] = elemento
registroDeporte[modalidad + '100m'] = 'Esta es… 100 metros mariposa'
registroDeporte[modalidad + '200m'] = 'Esta es… 200 metros mariposa'
```

El resultado sería de nuevo el mismo objeto de antes.

Esto quizás puede parecer una tontería, pero los que están acostumbrados a usar Javascript seguro que han tenido que hacer cosas rebuscadas para poder generar al vuelo propiedades en objetos con nombres computados. Es decir, de una manera cómoda nuestros objetos podrán tener propiedades de cualquier nombre que se necesite, aunque desconozca de antemano esos nombres y solo los pueda calcular en tiempo de ejecución.

Conclusión

Estas posibilidades de Javacript en ES6 nos facilitan bastante la escritura de código con objetos literales, ahorrando caracteres y permitiendo una lectura más rápida a los ojos del hombre. Merece la pena tenerlas en cuenta y usarlas siempre que podamos.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *18/10/2016*
Disponible online en *http://desarrolloweb.com/articulos/literales-objeto-es6.html*

Template Strings en ES6

Qué son las template strings, o cadenas de texto de plantilla, una alternativa de sintaxis Javascript

disponible en ES6 para creación de cadenas con interpolación de variables.

En este artículo vamos a hablar de una de las mejoras de ES6 aplicadas al lenguaje Javascript, las "Template Strings", que nos sirven para dulcificar la sintaxis en la creación de cadenas donde queremos embutir contenido de variables. Es una de las novedades de EcmaScript 2015, de las muchas que nos ofrece y que estamos revisando en el Manual de ES6.

Las template strings, o cadenas de texto de plantilla, son una de las herramientas de ES6 para trabajo con cadenas de caracteres que nos pueden venir muy bien para producir un código Javascript más claro. Usarlas es por tanto una recomendación dado que facilitará el mantenimiento de los programas, gracias a que su lectura será más sencilla con un simple vistazo del ojo humano.

La sintaxis es bien simple y muy fácil de entender, como veremos en este artículo.

Interpolación de variables en cadenas con Javascript tradicional

En un programa realizado en Javascript, y en cualquier lenguaje de programación en general, es normal crear cadenas en las que tenemos que juntar el contenido de literales de cadena con los valores tomados desde las variables. A eso le llamamos interpolar.

```
var sitioWeb = "DesarrolloWeb.com";
var mensaje = 'Bienvenido a ' + sitioWeb;
```

Eso es muy fácil de leer, pero a medida que el código se complica y en una cadena tenemos que interpolar el contenido de varias variables, el código comienza a ser más enrevesado.

```
var nombre = 'Miguel Angel';
var apellidos = 'Alvarez'
var profesion = 'desarrollador';
var perfil = '<b>' + nombre + ' ' + apellidos +  '</b> es ' + profesion;
```

Quizás estás acostumbrado a ver esto así. El código está bien y no tiene ningún problema, pero podría ser mucho más bonito si usas los template strings.

Crear un template string

Para crear un template string simplemente tienes que usar un carácter distinto como apertura y cierre de la cadena. Es el símbolo del acento grave.

```
var cadena = `Esto es un template String`;
```

> **Nota:** El acento grave no se usa en castellano pero sí en otros idiomas como catalán o portugués. Es un acento que va en inclinación opuesta al acento tradicional del español que es agudo.

Usos de las cadenas plantilla

Las cadenas de plantilla tienen varias características interesantes que, cómo decíamos, facilitan la sintaxis. Veremos a continuación algunos de ellos con código de ejemplo.

Interpolación de valores

Creo que lo más interesante es el caso de la interpolación que genera un código poco claro hasta el momento. Echa un vistazo al código siguiente que haría lo mismo que el que hemos visto anteriormente del perfil.

```
var nombre = 'Miguel Angel';
var apellidos = 'Alvarez'
var profesion = 'desarrollador';
var perfil = `<b>${nombre} ${apellidos}</b> es ${profesion}`;
```

Como puedes comprobar, dentro de un template string es posible colocar expresiones encerradas entre llaves y precediendo de un símbolo "[[--body--]]quot;. Algo como ${ expresion }.

En las expresiones podemos colocar código que queramos volcar, una vez evaluado, dentro de la cadena. Las usamos generalmente para colocar valores de varibles, pero también servirían para colocar operaciones matemáticas, por ejemplo.

```
var suma = `45 + 832 = ${45 + 832}`;
```

O bien algo como esto:

```
var operando1 = 7;
var operando2 = 98;
var multiplicacion = `La multiplicación entre ${operando1} y ${operando2} equivale a  ${operando1 * operando2}`;
```

Saltos de línea dentro de cadenas

Hasta ahora, si queremos hacer una cadena con un salto de línea teníamos que usar el caracter de escape "contrabarra n".

```
var textoLargo = "esto es un texto\ncon varias líneas";
```

Con un template string tenemos la alternativa de colocar el salto de línea tal cual en el código y no producirá ningún problema.

```
var textoLargo = `esto es un texto
con varias líneas`;
```

Conclusión

Esto es lo básico sobre las cadenas de plantilla, que deberías usar ya mismo en tu código, pues te permiten una sintaxis bastante mejorada en operaciones habituales con cadenas de caracteres.

Hay que advertir que todavía hay algunas otras cosas interesantes, detalles más avanzados pero también útiles que se podrían hacer con los template string, que no hemos visto en este artículo, pero que revisaremos más adelante.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *23/01/2017*
Disponible online en *http://desarrolloweb.com/articulos/template-strings-es6.html*

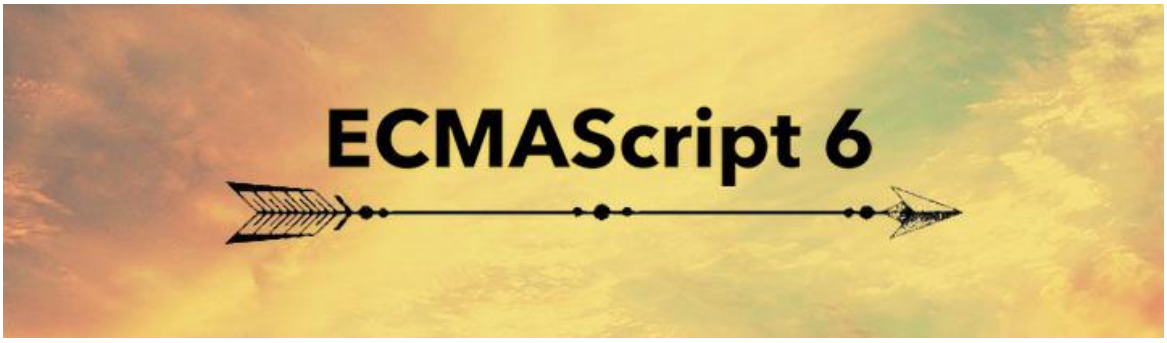
Arrow Functions en ES6

Qué son las funciones flecha, o arrow functions, disponibles en Javascript a partir de la actualización ES6 del lenguaje.

Las "arrow functions" de ES6 son una nueva manera de expresar las funciones de siempre, de un modo resumido y con algunas características nuevas que explicaremos en este artículo del Manual de ES6.

Aunque son comúnmente conocidas como arrow functions, también podrás oír hablar de ellas con su denominación en español, funciones flecha, o como "fat arrow functions", ya que para formar la flecha se usa una línea doble, del signo matemático igual "=".

Además de servir como azúcar sintáctico, son además una de las novedades más representativas de ES6, y que nos soluciona uno de los problemas más representativos y clásicos de Javascript en su versión ES5, el nuevo contexto generado por las funciones normales.



Cómo expresar una función con las arrow functions

Esta parte es bien simple, en vez de usar la palabra clave function se utiliza el símbolo de la flecha gorda, como se puede ver en el siguiente código:

```
let mifuncion = () => {
  //código de la función
}
```

Como has visto, no solo se usa la flecha, sino que los paréntesis donde se colocarían los parámetros de la función también se mueven de lado, colocándolos antes de la flecha.

La invocación de la función se realizaría como ya conoces.

```
mifuncion();
```

Parámetros de las funciones flecha

El tratamiento de los parámetros se realiza como hasta ahora, simplemente se colocan entre los paréntesis. Veamos este segundo ejemplo.

```
let saludo = (nombre, tratamiento) => {
  alert('Hola ' + tratamiento + ' ' + nombre)
}

//invocamos
saludo('Miguel', 'sr.');
```

El único detalle es que, en el caso que tengamos un único parámetro, podemos ahorrarnos colocar esos paréntesis.

```
let cuadrado = numero => {
  return numero * numero;
}
```

Ausencia de las llaves de la función

Existe otro caso especial, en el que podemos también ahorrarnos algún carácter extra, en este caso las llaves de apertura y cierre de la función. Sería cuando solamente tenemos una línea de código en nuestra función.

La función del saludo la podrías ver así también:

```
let saludo = (nombre, tratamiento) => alert('Hola ' + tratamiento + ' ' + nombre);
```

Ausencia de la palabra return

Si tenemos una única línea de código y la función devuelve un valor también nos podríamos ahorrar la palabra return, además de las llaves como se dijo antes.

La función del cuadrado de un número podría expresarse así.

```
let cuadrado = numero => numero * numero;
```

Código más compacto

Como ves, explotando todas las posibilidades de las funciones flecha, podemos obtener un código muy compacto. Para observar este hecho puedes compararlo con la declaración de una función de una manera tradicional en ES5:

```
var cuadrado = function(numero) {
  return numero * numero;
}
```

Quizás en la declaración de una función así sola en el código no se llegue a ver tanta ventaja, pero al tratarse Javascript de un lenguaje lleno de funciones callback se consigue bastante ahorro de líneas de código.

Mira este código asíncrono usando serTimeout():

```
setTimeout(function() {
  alert('me muestro pasado 1 segundo')
}, 1000);
```

Podrías verlo de esta manera usando arrow functions.

```
setTimeout( () => alert('Me muestro pasado 1 segundo'), 1000);
```

Pero todavía podemos captar más las ventajas de esta sintaxis ES6 en estructuras como las promesas. Como ya vimos en el artículo de promesas en ES6, se tienen que definir generalmente dos callbacks, uno para el

caso positivo y otro para el negativo. Expresadas esas funciones con arrow functions quedaría como esto:

```
funcionQueDevuelvePromesa()
  .then( valor => alert(valor) )
  .catch( error => alert(error) );
```

Evitar generar un nuevo contexto en this

Cuando usas funciones callback éstas generan un nuevo contexto sobre la variable "this". Es un efecto que si tienes experiencia con Javascript conocerás de sobra. En estos casos, para poder acceder al this anterior se hacían cosas como "var that = this", o quizás hayas usado el ".bind(this)" para bindear el contexto.

Por si no queda claro, mira el siguiente código:

```
var objTest = {
  retardo: function() {
    setTimeout(function() {
      this.hacerAlgo();
    }, 1000);
  },

  hacerAlgo: function() {
    alert('hice algo');
  }
}

objTest.retardo();
```

En la función setTimeout() estamos enviando un callback que genera un nuevo contexto, por tanto, no puedes acceder a this dentro de esa función. O mejor dicho, sí puedes acceder, pero no te devolverá lo que quizás se espere, que sería el propio objeto objTest. Es por eso que al ejecutar ese código te saldría un error:

```
Uncaught TypeError: this.hacerAlgo is not a function
```

Simplemente es que this ya no es el propio objeto y por tanto no existe el método que estás buscando.

La solución en ES5 pasaría por bindear this o cachear la referencia a this en una variable local que sí exista en el ámbito de la función callback. Realmente no importa mucho que veamos el código para resolver esto en ES5, ya que en ES6 y con las funciones flecha lo podríamos resolver de una manera mucho más elegante.

```
var objTest = {
  retardo: function() {
    setTimeout( () => {
      this.hacerAlgo();
    }, 1000);
  },
```

```
  hacerAlgo: function() {
    alert('hice algo');
  }
}

objTest.retardo();
```

Ahora la función enviada como callback a setTimeout() está definida con una arrow function y por tanto no genera contexto nuevo en la variable this. Es por ello que al intentar invocar a this.hacerAlgo() no generará ningún error y se ejecutará perfectamente ese método hacerAlgo().

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *28/03/2017*
Disponible online en *http://desarrolloweb.com/articulos/arrow-functions-es6.html*

Parámetros con valores predeterminados en funciones Javascript ES6

Conoce la nueva característica de Javascript ES6 que te permite definir valores predeterminados para los parámetros de las funciones.

En la mayoría de los lenguajes de programación se permite definir valores predeterminados a los parámetros recibidos en las funciones, por lo que esta nueva característica de ES6 no sorprenderá seguramente a la mayoría de los lectores. No obstante es algo nuevo en las últimas versiones de Javascript, por lo que vamos a analizar con detalle esta característica del lenguaje.

Básicamente, los valores por defecto en los parámetros permiten asignar de manera predeterminada datos, que se asignan a los argumentos de la función, en caso que no se indique ningún valor en su invocación. Esos datos predeterminados se indican en la declaración de la función y resultan muy cómodos, a la vez que añaden versatilidad a las funciones de Javascript. Veremos a continuación ejemplos con los que aclarar posibles dudas y estudiar el detalle de este comportamiento.

Nota: Por si aún te surgen dudas sobre por qué es importante esta novedad de ES6, puedes ver cómo se tenía que hacer a anteriormente este mismo comportamiento "a mano", ya que es algo que realmente resulta de mucha utilidad en el día a día de la programación y Javascript no lo soportaba. Sobrecarga de funciones y parámetros por defecto con ES5.

Definir parámetros con valores por defecto en Javascript

Igual que en otros lenguajes, para definir los valores predeterminados de los parámetros, realizamos una asignación de esos valores a los parámetros en la cabecera de la función

```
function saludar(nombre = 'Miguel Angel') {
  console.log('Hola ' + nombre);
}
```

Esta función recibe un parámetro llamado "nombre", con un valor predeterminado. Este valor se asignará en caso que al invocar a la función no le pasemos nada.

```
saludar();
```

Eso produciría la salida por consola "Hola Miguel Angel".

Pero, aunque indiquemos un valor predeterminado, podemos seguir invocando a la función enviando un valor diferente como parámetro.

```
saludar('DesarrolloWeb.com');
```

Justamente ésta es la versatilidad. La función sigue trabajando tal como lo conocíamos, agregando la posibilidad de tener un parámetro definido aunque no le pasemos nada.

El orden de los valores predeterminados a los parámetros importa

Los parámetros predeterminados en las funciones son muy fáciles de implementar, sin embargo tenemos que seguir unas pocas reglas. Básicamente, la más importante es la del orden de los parámetros en las llamadas a las funciones, que debe realizarse tal como está definido en la cabecera de la función. Esto es obvio, pero afecta directamente a los parámetros con valores por defecto en las funciones.

En resumen queremos hacer notar que, en una función donde unos argumentos tienen valores por defecto y otros no, debemos asegurarnos de colocar valores predeterminados en los últimos argumentos y no en los primeros. Creo que se ve mejor con un ejemplo.

Tenemos una función llamada "potencia" que calcula la potencia entre la base y el exponente. La base siempre la tenemos que indicar, pero sin embargo el exponente queremos que tenga un valor predeterminado. Si no indicamos nada, se entiende que el exponente será 2.

La función en cuestión es esta:

```
function potencia(base, exponente = 2) {
  let resultado = 1;
```

```
  for (let i = 0; i < exponente; i++) {
    resultado *= base;
  }
  return resultado;
}

console.log(potencia(3)); // indica 9 como resultado
console.log(potencia(3, 3)); // indica 27 como resultado
```

Fíjate que el parámetro que tiene valor por defecto es el último parámetro. Así te funcionará bien, puesto que si lo hiciéramos al revés no obtendríamos un resultado correcto.

Observa ahora este código:

```
function potenciaMal(exponente = 2, base) {
  let resultado = 1;
  for (let i = 0; i < exponente; i++) {
    resultado *= base;
  }
  return resultado;
}
```

En este caso tenemos un problema por haber colocado valor predeterminado en el primer parámetro y en el segundo no. Se apreciará si enviamos un único parámetro a la función, donde obtendremos resultados quizás inesperados:

```
console.log(potenciaMal(3)); // esto muestra NaN
```

El resultado ahora es "Not a Number" (NaN) porque el único valor indicado al invocar la función se asocia al primer parámetro (manda el orden de los parámetros en la declaración de la función, sin importar si existen o no valores predeterminados). El problema por tanto se nos da con el segundo argumento, que no tenía un valor predeterminado, y al no enviarlo en la invocación simplemente se queda indefinido. Al realizar los cálculos la función nos dice que el resultado no es un número (NaN).

Si aún no lo ves, puedes leer el código de esta función, que tiene el mismo problema.

```
function log(valor1 = 'Predeterminado', valor2) {
  console.log(valor1);
  console.log(valor2);
}
```

Si invocas la función enviando un solo parámetro, observarás el error:

```
log('algo'); // escribe 'algo' y luego 'undefined'
```

Usar variables para indicar los valores predeterminados

Algo interesante que podemos hacer para definir los valores predeterminados es usar variables. Por ejemplo:

```
let escuela = 'EscuelaIT';
function saludar(nombre = escuela) {
  console.log('Hola ' + nombre);
}
```

Ahora, si llamamos a saludar() sin enviar parámetros, dirá "Hola EscuelaIT".

Esto podría ser todavía más útil en un caso como el siguiente:

```
function potencia(base, exponente = base) {
  let resultado = 1;
  for (let i = 0; i < exponente; i++) {
    resultado *= base;
  }
  return resultado;
}
```

Como puedes observar, el valor por defecto indicado para el argumento "exponente" será el mismo valor que el indicado para "base". Así, indicada una base, se calculará la potencia cuyo exponente es la misma base.

```
console.log(potencia(1)); // muestra 1
console.log(potencia(2)); // muestra 4
console.log(potencia(3)); // muestra 27
console.log(potencia(4)); // muestra 256
console.log(potencia(4, 2)); // muestra 16
console.log(potencia(2, 1)); // muestra 2
```

Conclusión

Los valores predeterminados dan mucho juego. De hecho, en Javascript se venían usando, aunque había que programar de manera manual la función para que, en caso de no recibir parámetros se tomasen valores predeterminados, agregando líneas de código a la función, que a partir de ahora y gracias a ES6 se hacen innecesarias.

Puedes consultar otra cantidad de interesantes novedades del lenguaje en el Manual de Javascript con ECMAScript 2015.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *30/11/2017*
Disponible online en *http://desarrolloweb.com/articulos/parametros-predeterminados-funciones-es6.html*

Operador Rest, Javascript ES6

El operador Rest disponible en Javascript ES6 sirve para recibir cualquier número de parámetros en una función en forma de array.

Esta característica de ES6 nos ofrece otra vía de evitar trabajo repetitivo en las funciones con el tratamiento de parámetros. En este sentido ya vimos que ES6 ofrece la posibilidad de definir valores predeterminados a las funciones, pero ahora el objetivo es bien distinto. El operador rest en realidad sirve para obtener cualquier número de parámetros de una forma estructurada, mediante un array de valores.

En si "rest operator" se escribe mediante tres puntos seguidos, uno detrás de otro (...). Son como los puntos suspensivos de la escritura tradicional. Su funcionamiento es bastante sencillo, como veremos a continuación.

Uso elemental del operador rest

Podemos usar el operador rest en una función, dentro del juego de parámetros definido en su cabecera.

En el siguiente código puedes ver un primer ejemplo de uso, en el que indicamos "...numeros" como parámetros de la función.

```
function max(...numeros) {
  console.log(numeros);
}
```

Esto quiere decir que, cualquier cantidad de parámetros enviada a la hora de invocar a la función, se va a estructurar como un array y se va a conocer dentro de la función con el nombre de "numeros".

Puedes por tanto invocar a la función con 1 parámetro, 2, 3, 100 o incluso sin parámetros… Tanto da. Lo que tendrás siempre en el parámetro "numero" es un array, con 1, 2, 3, 100 o cero casillas.

Puedes comprobarlo al invocar a la anterior función con cualquier juego de parámetros.

```
max(1, 2, 6);
max();
max(1, 5, 6, 7, 10001);
```

Incluso podrías mezclar los tipos de los parámetros.

```
max("test", 4, true, 2000, "90");
```

Siempre te llega un array

Queremos insistir sobre este punto, pues la clave es que, al usar el operador rest, siempre te llega un array como valor del parámetro. Por ello al menos estas seguro que podrás usar los métodos conocidos para recorrido o manipulación del array.

Anteriormente podíamos también obtener cualquier número de parámetros en Javascript mediante el objeto arguments disponible en la función. La diferencia es que arguments no era un array y no permitía directamente ciertos tipos de operaciones, hasta que no lo convirtiéramos en un array mediante algún mecanismo.

Ejemplo de uso del operador rest

Ahora vamos a ver cómo definir bien la función max(), que se encargaría de devolver el valor máximo entre todos los parámetros recibidos en una función. Es decir, max() es una función que puede recibir cualquier número de parámetros y devuelve el valor máximo encontrado en ellos.

> **Nota:** para escribir esta función voy a tener en consideración dos cosas: 1) si no recibo parámetros, voy a devolver el valor cero (0). 2) voy a suponer que todos los parámetros serán valores numéricos.

Mi algoritmo realizará primero una comprobación para ver si no estoy recibiendo ningún parámetro. En ese caso devuelvo cero. Luego tomo como máximo el primer valor y lo voy comparando con el resto de los valores del array.

```
function max(...numeros) {
  if(numeros.length == 0) {
    return 0;
  }
  let max = numeros[0];
  for(let i = 1; i < numeros.length; i++) {
    if(numeros[i] > max) {
      max = numeros[i];
    }
  }
  return max;
}
```

> **Nota:** no quiero que pase desapercibido que estoy haciendo un recorrido con el for desde la segunda casilla del array (índice 1) hasta el final, ya que la primera casilla, (índice 0) ya la he tomado como

supuesto máximo para comenzar.

```
console.log(max()); // muesta 0
console.log(max(0, 10)); // muestra 10
console.log(max(0, -10)); // muestra 0
console.log(max(9, -22, 6)); // muestra 9
```

Tomar sólo los últimos parámetros con el operador rest

El operador rest nos ofrece en el array solamente los parámetros a los que no les hemos asignado un nombre. Es decir, puedo perfectamente en una función recibir un número de parámetros en variables normales y luego cualquier número de parámetros en un array con el operador rest.

Este comportamiento, por ejemplo, nos serviría para simplificar bastante el código de nuestra función max().

```
function max(max = 0, ...numeros) {
  for(let i = 0; i < numeros.length; i++) {
    if(numeros[i] > max) {
      max = numeros[i];
    }
  }
  return max;
}
```

Ha quedado bastante más compacto, gracias a que el primer parámetro, que vamos a tomar como supuesto máximo en el algoritmo, lo vamos a recibir de manera individual. Además le hemos asignado el valor predeterminado cero. El resto de parámetros será el array que recorreremos para ir comparando con el supuesto máximo.

Por si el ejemplo anterior no te queda suficientemente claro, tenemos un segundo ejemplo en el que vamos a recibir cualquier número de parámetros. Queremos saber si el primer parámetro se encuentra repetido en cualquiera de los parámetros indicados después.

```
function enLista(buscar, ...nombres) {
  for(let i = 0; i < nombres.length; i++) {
    if(buscar === nombres[i]) {
      return true;
    }
  }
  return false;
}

console.log(enLista('Miguel', 'Diego', 'Noe', 'Miguel')); // muestra true
console.log(enLista('Miguel', 'Diego', 'Noe', 'Manolo')); // muestra false
```

Conclusión

El operador Rest no es complicado y nos puede ahorrar código mucho menos legible, y más largo, usado anteriormente para obtener cualquier número de argumentos en una función.

Puedes obtener muchas otras novedades de Javascript en el Manual de ES6.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *19/12/2017*
Disponible online en *http://desarrolloweb.com/articulos/operador-rest-javascript-es6.html*

Operador spread (de propagación) en Javascript ES6

Qué es el operador de propagación en ES6, la nueva versión de Javascript, conocido también como spread operator.

En este artículo vamos a conocer un nuevo operador de Javascript, que nos oferce la versión de ECMAScript 2016, ES6, ya disponible en todos los navegadores modernos. Se trata del "spread operator" u operador de propagación, que nos puede ayudar a ahorrar unas cuantas líneas de código en muchas ocasiones, así como crear un código más consistente.

Para los que conocieron el operador rest, en el pasado artículo del manual de Javascript ES6, cabe decir que es más o menos la operación contraria. Mientras que el operador rest genera un array a partir de una lista de valores, el operador spread genera una lista de valores a partir de un array.

Además, estos operadores tienen en común entre sí el utilizar exactamente la misma sintaxis de los tres puntos seguidos. Entonces, si el Rest operator es "..." y es exactamente el mismo código que usamos para el spread operator "..." ¿Cómo lo distinguimos? La diferencia está en que el rest operator lo usas en la cabecera de una función, al implementarla y el spread operator lo usas en la invocación. Lo veremos mejor con un ejemplo.

Spread operator en funcionamiento

Para nuestro ejemplo vamos a hacer uso de la función min() de la clase Math de Javascript. Por si no lo sabes, esta función espera recibir cualquier número de parámetros numéricos y devuelve el que tenga el valor menor.

Por ejemplo, podemos invocarla así.

```
Math.min(2, 5, 7, 1, 9);
```

Esto nos devolverá el valor 1, que es el mínimo de los parámetros enviados.

Para hacer uso del spread operator necesitamos partir de un array y conseguiremos convertir ese array en una lista de parámetros, de la siguiente manera.

```
let miArray = [2, 5, 7, 1, 9];
let minimo =Math.min(...miArray);
```

Como puedes ver, el operador de propagación se idéntico al operador rest. Lo único que aquí lo estamos usando en la invocación de una función.

Uso del operador rest en combinación con el operador spread

Estos dos operadores muchas veces los encontramos de la mano, usados en el mismo ejemplo.

Por ejemplo, pongamos una función que recibe cualquier número de cadenas por parámetro y nos devuelve un array con esas mismas cadenas, pero eliminados los espacios en blanco antes y después de la cadena.

Comencemos por refrescar la memoria, viendo la aplicabilidad del operador Rest en este ejemplo.

```
function limpiarEspacios(...cadenas) {
  for (let i=0; i<cadenas.length; i++) {
    cadenas[i] = cadenas[i].trim();
  }
  return cadenas;
}
```

En este caso hemos usado el operador "..." en la invocación de una función, luego lo que quiere decir es que una lista de parámetros la convertirá en un array.

Podemos ver el ejemplo en marcha con este código.

```
let cadenasLimpias = limpiarEspacios('hola ', ' algo ', ' más');
console.log(cadenasLimpias);
```

Ejecutando ese código observarás que "cadenasLimpias" es un array de cadenas, eliminadas los espacios.

Pero ¿Qué pasa si las cadenas que queremos limpiar están en un array?

```
const cadenasOriginales = ['hola ', ' algo ', ' más'];
let cadenasLimpias = limpiarEspacios(cadenasOriginales);
```

Esto te arrojaría un error, pues dentro de la función se intentará hacer trim() sobre un array y Javascript te diría que "cadenas[i].trim is not a function".

Bien, aquí es donde el operador spread sale al rescate, permitiendo convertir tal array en una lista de parámetros para usar en la invocación de la función.

```
const cadenasOriginales = ['hola ', ' algo ', ' más'];
let cadenasLimpias = limpiarEspacios(...cadenasOriginales);
```

En este caso todo irá bien, ya que el array se convertirá en una lista de parámetros, justo lo que necesita la función limpiarEspacios().

Nuevamente queremos insistir en que el operador es exactamente el mismo, es decir, usa la misma sintaxis. Solo que su función será diferente si la usas en la definición de una función o en su invocación. Al definir una función se llama "rest operator" y al invocarla se llama "spread operator".

Otros usos de spread operator

El operador de propagación también se puede usar al definir literales de array en base a otros arrays. Por ejemplo, mira el código siguiente.

```
let misConocimientos = ['variables', 'operadores', 'estructuras de control', 'funciones'];
let aprendido = ['rest operator', 'spread operator'];
let misConocimientosAmpliados = [...misConocimientos, ...aprendido, 'otra cosa más'];
```

Para la creación del array "misConocimientosAmpliados, definido como un literal, estamos usando dos veces el operador de propagación. Una para incorporar todos los valores de "misConocimientos", como una lista, y otra para incorporar como otra lista de valores el array "aprendido".

Conclusión

Esperamos que estas explicaciones sobre el operador spread, también llamado operador de propagación, te sirvan para incorporar nuevas alternativas y mejoras en tu código Javascript. Son muy útiles ya que nos permiten realizar de una manera muy cómoda cosas que antes de ES6 debeíamos hacer usando más líneas de código.

Aunque comparta sintaxis con el operador rest de ES6, esperamos que no te lies al usarlo y puedas distinguir qué hace cada cual. Rest sirve para convertir una lista en un array y spread para hacer el paso contrario, a partir de un array generar una lista.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *11/01/2018*
Disponible online en *http://desarrolloweb.com/articulos/operador-spread-es6.html*

ES6 Modules

Conoce y aprende a usar los módulos de ES6, ya soportados de manera nativa por los navegadores. Uso de import y export en los ES6 Modules.

Una de las estupendas novedades de ES6 es la posibilidad de crear módulos, que son piezas de código que podemos escribir en ficheros independientes. Los módulos pueden tener código, como clases, funciones, objetos o simples datos primitivos, que se puede importar desde otros archivos.

Con la palabra "export" consigues exponer algún miembro del módulo, para que se pueda usar desde fuera. Con la palabra "import" consigues traerte algo exportado por un módulo independiente.

Aunque lo cierto es que seguro que muchos de vosotros ya venís usando módulos desde hace años, apoyándose en transpiladores como Babel, la novedad es que hoy ya podemos usarlos en la mayoría de los navegadores. Safari, Chrome, Opera ya los pueden usar de manera nativa y Firefox y Edge también los tienen implementados, aunque todavía hay que activarlos en la configuración del navegador (aunque posiblemente cuando leas este artículo también los puedas usar sin necesidad de configurar nada).

En este artículo te explicaremos cómo usar "ES6 modules" y como exportar e importar cosas entre distintos módulos.

> **Nota:** Si quieres conocer otras características del estándar de Javascript, te aconsejamos leer el Manual de ECMAScript 2015 (ES6).

Cómo usar un módulo desde un archivo HTML

Lo primero que hay que decir es que debes hacer una acción especial para que el navegador procese correctamente los archivos Javascript que usan módulos. Básicamente se trata de advertir al navegador que el script Javascript usa módulos, para lo que necesitas traerte ese código desde el HTML usando una sintaxis especial en la etiqueta SCRIPT. Simplemente tenemos que marcar con type="module" el script que queremos incluir.

```
<script src="index.js" type="module"></script>
```

A partir de este momento, en index.js ya puedes usar imports de otros módulos, tal como explicaremos

seguidamente.

Compatibilidad en navegadores antiguos

El uso de este "type" es importante, no solo porque así te aseguras que puedes usar ES6 modules, sino porque puedes implementar alternativas para diversos navegadores. Aunque el soporte es nativo en las recientes versiones de browsers, es obvio que los módulos no van a funcionar en todos los navegadores viejos o poco actualizados (léase los IE antiguos). Para ellos podemos disponer de una alternativa.

De momento, debes percibir que no todos los navegadores entenderán qué es un type="module". Al no saber qué tipo de script es un "module", los navegadores antiguos simplemente no harán nada con él. Es lógico, pues aunque lo abrieran no lo podrían entender.

Para los navegadores viejos hay que seguir transpilando, con Babel o similares y creando alternativas de scripts que no usen módulos ES6.

> **Nota:** puedes obtener más información de la transpilación y Babel en este artículo: Introducción a ES6.

Los archivos transpilados a ES5, sin usar módulos los cargarás por medio de una etiqueta SCRIPT que incluya un atributo "nomodule", de esta manera:

```
<script src="codigo-transpilado.js" nomodule></script>
```

En resumen:

- **Los navegadores modernos** entenderán el atributo "nomodule", por lo tanto no accederán a este script, porque sabrán que es un script pensado por y para los navegadores obsoletos.
- **Los navegadores antiguos** no entenderán el atributo "nomodule", pero tampoco le harán caso y cargarán este script alternativo.

> **Nota:** En este momento (diciembre 2017) Firefox ya acepta los módulos de ES6, aunque hay que activar en el navegador todavía su compatibilidad. Para hacerlo se debe entrar dentro de "about:config" (escribe ese mismo texto: "about:config" sin las comillas) y luego hay que activar a true la preferencia dom.moduleScripts.enabled.

Export

Como hemos dicho, usamos la sentencia export para permitir que otros módulos usen código del presente módulo.

Con un export puedes exportar todo tipo de piezas de software, como datos en variables de tipos primitivos, funciones, objetos, clases. Ahora vamos a comenzar viendo un caso sencillo de hacer un export,

luego veremos más alternativas.

```
export const pi = 3.1416;
```

Simplemente anteponemos la palabra export a aquello que queremos exportar hacia afuera.

Import

En el momento que queramos cargar alguna cosa de un módulo externo, usaremos la sentencia import. Para ello tenemos que indicar qué es lo que queremos importar y la ruta donde está el módulo que contiene aquello que se desea importar.

```
import { pi } from './pi-module.js';
```

Así estamos importando la anterior constante definida "pi", que estaba en un archivo aparte, en un módulo llamado "pi-module.js". Observarás por la ruta que pi-module.js está en la misma ruta que el archivo desde el que importamos.

Una vez importado ese dato (lo que sea que se importe) lo podemos usar como si la declaración hubiera sido hecha en este mismo archivo.

Resumen de uso de módulos

Para que te quede claro el proceso, vamos a ver listados los tres archivos que puedes tener para hacer el primer test a los módulos de ES6.

########## index.html

Tendremos un archivo index.html que tendrá que acceder al script que trabaja con módulos.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>test es6 modules</title>
</head>
<body>
  <h1>Test de los módulos ES6</h1>

  <script src="index.js" type="module"></script>
</body>
</html>
```

########## index.js

Este es el archivo principal de Javascript, en el que vamos a hacer los import.

Una vez que has hecho el import correspondiente, puedes usar aquello que hayas importado.

```
import { pi } from './pi-module.js';
console.log(pi);
```

Podrías importar varias cosas al mismo tiempo de un módulo, en cuyo caso entre las llaves colocarías todos los elementos a importar, separados por comas.

```
import { pi, e, log2 } from './pi-module.js';
```

########## pi-module.js

Este es el archivo que realiza el export, el módulo que expone hacia afuera las piezas de software que sean necesarias.

```
export const pi = 3.1416;

// En este archivo puedo tener código Javascript que no exporto... es como código privado del módulo
var variableLocalPrivada = 222;
```

Es interesante observar cómo el módulo puede tener cosas que exporta y cosas que se quedan dentro. Por ejemplo puedes tener funciones que realizan cálculos internos, que no necesitan o que no deberían ser invocadas desde fuera del módulo. Obviamente, te da el trabajo de ser explícito en todo lo que deseas exportar, pero te permite tener código privado que no se puede usar desde fuera.

Export default

Cuando exportamos algo en un módulo podemos definir que sea la exportación por defecto. No es necesario que se exporte nada por defecto en un módulo, pero en caso de hacerlo, debes tener en cuenta que sólo se puede exportar una cosa "default" por módulo.

Por ejemplo, así se puede exportar una función en un módulo, marcando que sea la exportación predeterminada:

Archivo validate-email-function.js con export default

```
function validateEmail(email) {
  if(email.indexOf('@') != -1) {
    return true;
  }
  return false
}

export default validateEmail
```

Importar el elemento default

A la hora de importar aquel elemento marcado como default tenemos que hacer el import de una manera ligeramente diferente.

```
import validateEmail from './validate-email-function.js';
```

Crear un alias (namespace) para los elementos importados

Otra cosa útil que puedes hacer es crear un alias para los elementos importados, con la palabra "as". Vamos a verlo con un ejemplo.

Imagina que tienes varias funciones definidas en un módulo.

```
export function reir() {
  console.log('jajaja');
}

export function reirFuerte() {
  console.log('JAJAJAJAJA');
}

export function reirSuave() {
  console.log('jeje');
}
```

Ya sabes que podrías importarlas todas de una vez, separando por comas todo aquello que quieres importar. Luego podrás usar las funciones como si las hubieras declarado en este mismo archivo.

```
import { reir, reirFuerte, reirSuave} from './risas.js';

reir();
reirFuerte();
reirSuave();
```

Pero otra posibilidad sería traernos todos los elementos exportados, asignando un alias, como un espacio de nombres, mediante el cual se conocerán en el módulo que importa. En este caso, todas las funciones dependerán del alias, como puedes ver a continuación.

```
import * as risas from './risas.js';

risas.reir();
risas.reirFuerte();
risas.reirSuave();
```

Organizar los exports, para declararlos todos de una vez

Algo que puedes hacer es exportar una única vez en el módulo todo lo que quieras que se vea desde fuera. Es una práctica normal, que permite tener un poco más de orden en lo que se está entregando en un módulo hacia el exterior. Obviamente, es totalmente opcional.

```
function titular(cadena) {
  return `<h1>${cadena}</h1>`;
}
function parrafo(cadena) {
  return `<p>${cadena}</p>`;
}
function salto() {
  return '<br>';
}

export const tags = {
  titular,
  parrafo,
  salto
}
```

Lo único que estás exportando es un objeto, donde tienes las funciones que deseas exponer hacia fuera. Ahora, al importar, con que hagas un import, podrás traerte todos los elementos exportados. Aunque, como ves, realmente es solo uno que los contiene a todos.

```
import { tags } from './tags.js';

console.log(tags.titular('Hola!'));
console.log(tags.parrafo('Esto es un test de los módulos ES6'));
```

Conclusión sobre los módulos ECMAScript 2015

Con lo que has aprendido no deberías tener problemas al usar módulos, definidos en el estándar ECMAScript 2015 (ES6). Como has visto, permite organizar el código de distintas maneras, colocando cada cosa en su sitio: cada módulo con su responsabilidad definida.

Lo más interesante es que ya se puede usar en los navegadores modernos, por lo que es una realidad ya disponible para la programación frontend. La pena es que los navegadores antiguos no la van a conocer nunca, por lo que hasta que desaparezcan finalmente tenemos que seguir usando transpiladores y compactando el código en un único fichero. No es tarea difícil y hemos aprendido a colocar scripts diferentes para navegadores que aceptan módulos o no, pero significa un paso adicional, que lógicamente deberías automatizar para que no suponga un esfuerzo constantemente.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *03/10/2017*
Disponible online en *http://desarrolloweb.com/articulos/es6-modules.html*

Promesas en ES6

Una de las características más interesantes de ECMAScript 2015 es la incorporación de las promesas, que nos permiten gestionar de una manera más clara las funciones asíncronas (aquellas que tardan un tiempo indeterminado en ejecutarse). En Javascript, como lenguaje no bloqueante ante código asíncrono, se definen funciones callback a ejecutar cuando se acaban los procesos asíncronos. Mediante las promesas las funciones callback se pueden organizar de una manera muy potente y sencilla.

Introducción a las Promesas de ES6

En este artículo te vamos a explicar qué son las promesas, como usarlas y cómo crear funciones que implementan promesas.

Las promesas son herramientas de los lenguajes de programación que nos sirven para gestionar situaciones futuras en el flujo de ejecución de un programa. Aunque es un concepto que usamos en Javascript desde hace un relativamente corto espacio de tiempo, ya se viene implementando en el mundo de la programación desde la década de los 70.

Las promesas se originaron en el ámbito de la programación funcional, aunque diversos paradigmas las han incorporado, generalmente para gestionar la programación asíncrona. En resumen, nos permiten definir cómo se tratará un dato que sólo estará disponible en un futuro, especificando qué se realizará con ese dato más adelante.

Aunque en Javascript se introducen en el estándar en ES6, lo cierto es que se vienen usando desde hace tiempo, ya que varias librerías las habían implementado para solucionar sus necesidades de una manera más elegante, e incluso existen bibliotecas independientes en Javascript que tenían como único propósito facilitar la creación de promesas, antes que el propio Javascript "Vanilla" las incorporarse.

Ahora con ES6 podemos beneficiarnos de sus ventajas a la hora de escribir un código más limpio y claro. De hecho son una de las novedades más destacadas de ES6.

Cómo usar promesas

Creo que para comenzar a entender las promesas nos viene muy bien comenzar viendo cómo podemos

gestionarlas, es decir, qué se debe hacer cuando ejecutamos funciones que nos devuelven promesas. Luego aprenderemos a crear funciones que implementan promesas y veremos que es también bastante fácil.

Por ejemplo, la función set() de Firebase, para guardar datos en la base de datos en tiempo real, devuelve una promesa cuando se realiza una operación de escritura.

> **Nota:** Da igual que no conozcas Firebase, y aunque puedes aprender en el Manual de Firebase, no es necesario para entender las promesas. Simplemente quiero que se entienda cómo gestionar una promesa. Olvida que esta función pertenece al API de Firebase, concéntrate en el esquema de trabajo de promesas, que es siempre el mismo.

Tiene sentido que se use una promesa porque, aunque Firebase es realmente rápido, siempre va a existir un espacio de tiempo entre que solicitamos realizar una escritura de un dato y que ese dato se escribe realmente en la base de datos. Además, la escritura podría dar algún tipo de problema y por tanto producirse un error de ejecución, que también deberíamos gestionar. Todas esas situaciones se pueden implementar por medio de dos métodos:

- then: usado para indicar qué hacer en caso que la promesa se haya ejecutado con éxito.
- catch: usado para indicar qué hacer en caso que durante la ejecución de la operación se ha producido un error.

Ambos métodos debemos usarlos pasándoles la función callback a ejecutar en cada una de esas posibilidades.

```
referenciaFirebase.set(data)
  .then(function(){
    console.log('el dato se ha escrito correctamente');
  })
  .catch(function(err) {
    console.log('hemos detectado un error', err');
  });
```

Fíjate que "referenciaFirebase.set(data)" nos devuelve una promesa. Sobre esa promesa encadenamos dos métodos, then() y catch(). Esos dos métodos son para gestionar el futuro estado de la escritura en Firebase y están encadenados a la promesa. Por si acaso no se entiende eso, podríamos leer este mismo código de esta manera.

```
referenciaFirebase.set(data).then(function(){
  console.log('el dato se ha escrito correctamente');
}).catch(function(err) {
  console.log('hemos detectado un error', err');
});
```

Igual así se ve mejor qué es lo que me refiero cuando digo que están encadenados. Insisto en esto para que nos demos cuenta que los then() y catch() forman parte de la misma cosa (la promesa devuela por el métood

set() de Firebase) y porque encadenar promesas es algo bastante normal y así nos vamos familiarizando mejor con cosas que usaremos en un futuro próximo.

> **Nota:** Estoy escribiendo código más parecido a ES5, porque así nos quedamos solo con la parte nueva, las promesas y no nos despistamos con sintaxis que quizás todavía no tienes perfectamente asimilada como las arrow functions. Pero obviamente, esas funciones callback enviadas a then() y catch() podrían ser perfectamente expresadas con arrow functions de ES6.

Otro detalle que no debe pasar desapercibido es que la promesa puede devolver datos. Es muy normal que esto ocurra. Por ejemplo queremos recibir algo de una base de datos y cuando la promesa se ejecuta correctamente querremos que nos llege ese dato buscado. No es el caso en este método set() de Firebase, porque una operación de escritura no te devuelve nada en esta base de datos, pero insisto que es algo bastante común.

En el caso que la promesa te devuelva un dato, lo podrás recibir como parámetro en la función callback que estás adjuntando al then().

```
funcionQueDevuelvePromesa()
  .then( function(datoProcesado){
    //hacer algo con el datoProcesado
  })
```

> **Nota:** Como estás viendo, al usar una promesa no estoy obligado a escribir la parte del catch, para procesar un posible error, ni tan siquiera la parte del then, para el caso positivo.

En el caso negativo implementado mediante el catch() siempre vamos a recibir un dato, que es el error que se ha producido al ejecutar la promesa y el causante de estar procesándose el correspondiente catch. Volviendo al ejemplo de antes, método set(), observa que el error lo hemos recibido en el parámetro de la función callback indicada en el catch().

Pirámide de callbacks

Seguro que habrás oído hablar del código spaguetti. Uno de los síntomas en Javascript de ello es lo que se conoce como pirámide de callbacks o "callback hell". Hay mucha literatura y ejemplos en Javascript sobre ello. Ocurre cuando quieres hacer una operación asíncrona, a la que le colocas un callback para continuar tu ejecución. Luego quieres encadenar una nueva operación cuando acaba la anterior y otra nueva cuando acaba ésta.

El método setTimeout() de toda la vida en Javascript nos sirve para escribir algo de código spaguetti y ver la temida pirámide de callbacks.

```
setTimeout(function() {
```

```
  console.log('hago algo');
  setTimeout(function() {
    console.log('hago algo 2');
    setTimeout(function() {
      console.log('hago algo 3');
      setTimeout(function() {
        console.log('hago algo 4');
      }, 1000)
    }, 1000)
  }, 1000)
}, 1000);
```

En resumen lo que hacemos es encadenar una serie de tareas, para realizarlas secuencialmente, una cuando acaba la otra. Funciona, pero ese código es un infierno para mantener, pues tiene difícil lectura y cuesta meterle mano para implementar nuevas funcionalidades. Las promesas nos pueden ayudar a mejorarlo, pero primero vamos a tener que aprender a implementarlas nosotros mismos.

Implementar una promesa

Ahora viene la parte interesante, en la que aprendemos a crear nuestras propias funciones que devuelven promesas. Esto se consigue mediante la creación de un nuevo objeto "Promise", como veremos a continuación. Pero antes de ponernos con ello debes tener bien claro el objetivo de una promesa: "hacer algo que dura un tiempo y luego tener la capacidad de informar sobre posibles casos de éxito y de fracaso"

Ahora verás el código y aunque pueda parecer confuso al principio, la experiencia usando promesas te lo irá clarificando naturalmente. Ten en cuenta que para crear un objeto "Promise" voy a tener que entregarle una función, la encargada de realizar ese procesamiento que va a tardar algo de tiempo. En esa función debo ser capaz de procesar casos de éxito y fracaso y para ello recibo como parámetros dos funciones:

- La función "resolve": la ejecutamos cuando queremos finalizar la promesa con éxito.
- La función "reject": la ejecutamos cuando queremos finalizar una promesa informando de un caso de fracaso.

```
function hacerAlgoPromesa() {
  return new Promise( function(resolve, reject){
    console.log('hacer algo que ocupa un tiempo...');
    setTimeout(resolve, 1000);
  })
}
```

Como puedes ver, nuestra función hacerAlgoPromesa() devolverá siempre una promesa (return new Promise). Se encarga de hacer alguna cosa, y luego ejecutará el método resolve (1 segundo después, gracias al serTimeout).

> **Nota:** Aun no estamos controlando posibles casos de fracaso, pero de momento está bien para no liarnos demasiado.

Esa misma función algunos programadores la preferirían ver escrita de este otro modo.

```
function hacerAlgoPromesa(tarea) {
  function haciendoalgo(resolve, reject) {
    console.log('hacer algo que ocupa un tiempo...');
    setTimeout(resolve, 1000);
  }
  return new Promise( haciendoalgo );
}
```

Es exactamente lo mismo que teníamos antes, solo que se ha ordenado el código de otra manera. Usa la alternativa que veas más clara.

Ahora vamos a ver cómo ejecutar esta función que nos devuelve una promesa, aunque si entendiste el principio del artículo ya lo tendrás bastante claro.

```
hacerAlgoPromesa()
  .then( function() {
    console.log('la promesa terminó.');
  })
```

Encadenar promesas

Como colofón a esta introducción a las promesas de ES6 queremos ver cómo nos facilitan la vida, creando un código mucho más limpio y entendible que la famosa pirámide de callbacks que hemos visto en un punto anterior. Si no estás familiarizado con el tema estoy seguro que te sorprenderás. Para que sea así, vamos directamente con el código fuente:

Imagina que quieres hacer algo y repetirlo por cuatro veces, ejecutando la función hacerAlgoPromesa() repetidas veces, de manera secuencial, una después de la otra.

```
hacerAlgoPromesa()
.then( hacerAlgoPromesa )
.then( hacerAlgoPromesa )
.then( hacerAlgoPromesa )
```

Eso, comparado con el spaguetti code de antes, tiene su diferencia ¿no? y es básicamente lo mismo, ejecutar una acción 4 veces con un retardo entre ellas.

> **Nota:** Obviamente en la realidad generalmente no repites lo mismo cuatro veces la misma operación, aunque podría ser, sino que puedes encadenar cuatro promesas distintas, una detrás de la otra, para realizar varias tareas diferentes.

Conclusión

A partir de aquí queda todavía por abordar diferentes puntos interesantes y útiles, como controlar posibles casos de error e informar de ellos en nuestras promesas, o poder ejecutar varias promesas en paralelo, en vez de secuencialmente. Todo eso lo iremos tratando, aunque espero que con este artículo se te abra un poco de luz y que puedas apreciar algunas de las ventajas de usar promesas ES6.

Ejemplo adicional de promesas en Javascript ECMAScript 2015

Si quieres investigar algo más sobre encadenar promesas, piensa que a veces a las funciones que devuelven promesas les tienes que pasar parámetros. ¿Cómo escribirías el chaining de promises? Para ser más claros, echa un vistazo a esta promesa.

```
function hacerAlgoPromesa2(tarea) {
  function haciendoalgo(resolve, reject) {
    console.log('Hacer ' + tarea + ' que ocupa un tiempo...');
    setTimeout(resolve, 1000);
  }
  return new Promise( haciendoalgo );
}
```

Es casi casi lo mismo que teníamos antes, solo que ahora le podemos pasar la tarea que quieres realizar. Lo que queremos es encadenar cuatro tareas diferentes, para ejecutar en secuencial, igual que antes. Pero tienes que pasarles parámetros distintos.

La solución la encuentras en el siguiente pedazo de código.

```
hacerAlgoPromesa('documentar un tema')
.then(function() {
  return hacerAlgoPromesa('escribir el artículo')
})
.then(function() {
  return hacerAlgoPromesa('publicar en desarrolloweb.com')
})
.then(function() {
  return hacerAlgoPromesa('recibir vuestro apoyo cuando compartís en vuestras redes sociales')
})
```

Échale un vistazo y trata de entenderlo. La clave es que para encadenar promesas la función a ejecutar como callback debe devolver también una nueva promesa.

Nos hemos quitado de en medio la pirámide de callbacks, ero tampoco creas que sería el mejor código para resolver este problema. Ya que estamos en un Manual de ES6, no queremos perder la oportunidad de mostrar un ejemplo del azúcar sintáctico que nos ofrecen las Arrow Functions.

Este código sería equivalente al anterior:

```
hacerAlgoPromesa('documentar un tema')
.then(() => hacerAlgoPromesa('escribir el artículo'))
.then(() => hacerAlgoPromesa('publicar en desarrolloweb.com'))
```

```
.then(() => hacerAlgoPromesa('...compartís en vuestras redes sociales'))
```

Mucho más limpio, no?

Como has visto, las promesas ayudan verdaderamente a organizar nuestro código. Pero todavía hay bastantes cosas que debes aprender sobre ellas. Te recomendamos seguir la lectura en este artículo que trata sobre las funciones resolve y reject usadas al implementar nuestras propias promesas.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *22/09/2016*
Disponible online en *http://desarrolloweb.com/articulos/introduccion-promesas-es6.html*

Implementar promesas: resolve / reject

Artículo práctico para explicar un código de implementación de promesas ECMAScript 2015, y las funciones resolve y reject.

En una entrega anterior del Manual de ES6 estuvimos hablando de las promesas y explicamos cómo nos pueden ayudar a organizar nuestro código. Vimos que en el lenguaje Javascript es fácil caer en lo que se conoce como "código spaguetti" y que las promesas nos ofrecen una vía excelente para organizar nuestro código evitando la pirámide de callbacks. Puedes consultar la Introducción a las promesas de ES6 para más información.

En esta ocasión vamos a ampliar la información en lo referente a la implementación de una función que devuelve una promesa, tratando tanto los casos positivos (éxito) como los casos negativos (fracaso). Además ahondaremos en otra información importante de lenguaje Javascript y la nueva API de acceso a recursos Ajax: fetch.

Nota: Si no conoces fetch también te recomendamos estudiarlo antes de abordar este artículo, en el artículo de introducción a fetch.

Funciones resolve y reject

Cuando implementamos una función que devuelve una promesa tenemos a nuestra disposición dos

funciones que permiten devolver el control al código que invocó la promesa. Esas funciones devuelven datos cuando la promesa se ejecutó normalmente y produjo los resultados esperados y cuando la promesa produjo un error o no pudo alcanzar los resultados deseados.

> **Nota:** En el artículo de Introducción a las promesas de ES6 ya vimos cómo usar el resolve para devolver el control en el caso positivo, pero no hemos visto cómo usar el reject para informar de un caso negativo.

El código de una función que devuelve una promesa tiene una forma inicial como esta:

```
function devuelvePromesa() {
  return new Promise( (resolve, reject) => {
        //realizamos nuestra operativa…
  })
}
```

Como puedes ver, este código devuelve una nueva promesa. Para crear la promesa usamos "new Promise". El constructor de la promesa lo alimentamos con una función que recibe dos parámetros: resolve y reject, que son a su vez funciones que podremos usar para devolver valores tanto en el caso positivo como en el caso negativo.

> **Nota:** Obviamente esos parámetros los puedes llamar como te apetezca, ya que son solo parámetros, pero los nombres indicados ya nos indican para qué sirve cada uno.

Ahora veamos nuestra operativa, que podría tener una forma como esta:

```
function devuelvePromesa() {
  return new Promise( (resolve, reject) => {
    setTimeout(() => {
      let todoCorrecto = true;
      if (todoCorrecto) {
        resolve('Todo ha ido bien');
      } else {
        reject('Algo ha fallado)
      }
    }, 2000)
  })
}
```

En nuestro código realizaremos cualquier tipo de proceso, generalmente asíncrono, por lo que tendrá un tiempo de ejecución durante el cual se devolverá el control por medio de una función callback. En la función callback podremos saber si aquel proceso se produjo de manera correcta o no. Si fue correcto usaremos la función resolve(), mandando de vuelta como parámetro el valor que se haya conseguido como

resultado. Si algo falló usaremos la función reject(), enviando el motivo del error.

Podremos usar esa función que devuelve la promesa con un código como este:

```
devuelvePromesa()
  .then( respuesta => console.log(respuesta) )
  .catch( error => console.log(error) )
```

Como ya supondrás, then() recibe la respuesta indicada en el resolve() y catch() el error indicado en el reject.

Cómo implementar una conexión Ajax con fetch que devuelve un texto

En el artículo de fetch vimos que es un nuevo modelo de trabajo con Ajax, pero usando promesas. Vimos que un fetch() te devuelve una respuesta del servidor, con datos sobre la solicitud HTTP, pero si lo que queríamos es acceder al texto de la respuesta, necesitábamos encadenar una segunda promesa, llamando al método text() sobre la respuesta. Esa segunda promesa nos complicó un poco el código de algo tan sencillo como es: dada una URL recibir el texto que hay en el recurso.

A continuación vamos a poner un código de alternativa, que realiza ambas promesas y directamente nos devuelve el texto de respuesta. Como es un código asíncrono, que tardará un poco en ejecutarse y después de ello debe devolver el control al script original, lo implementaremos por medio de una promesa.

```
function obtenerTexto(url) {
  return new Promise( (resolve, reject) => {
    fetch(url)
      .then(response => {
        if(response.ok) {
          return response.text();
        }
        reject('No se ha podido acceder a ese recurso. Status: ' + response.status);
      })
      .then( texto => resolve(texto) )
      .catch (err => reject(err) );
  });
}
```

Este código usa el modelo de encadenado de promesas, ejecutando operaciones asíncronas, una cuando termina la anterior.

Comenzamos haciendo el fetch() a una URL que recibimos por parámetro. Ese fetch devuelve una promesa, que cuando se ejecuta correctamente nos entrega la respuesta del servidor.

Si la respuesta estuvo bien (response.ok es true) entonces devuelve una nueva promesa entregada por la ejecución de la función response.text(). Si la respuesta no estuvo bien, entonces rechazamos la promesa con reject(), indicando el motivo por el que estamos rechazando con un error.

A su vez el código de response.text(), que devolvía otra promesa, puede dar un caso de éxito o uno de error. El caso de éxito lo tratamos con el segundo then(), en el que aceptamos la promesa con el resolve.

Tanto para el caso de error de fetch() como para un caso de error en el método text(), como para cualquier otro error detectado (incluso un código mal escrito) realizamos el correspondiente catch(), rechazando nuestra promesa original.

> **Nota:** Observa que un solo catch() te sirve para detectar cualquier tipo de error, tanto en la primera promesa (fetch) como en la segunda (text).

Este código lo puedes usar de la siguiente manera. Verás que tenemos un único método que nos devuelve directamente el texto.

```
obtenerTexto('test.txt')
  .then( texto => console.log(texto) )
  .catch( err => console.log('ERROR', err) )
```

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *06/06/2017*
Disponible online en *http://desarrolloweb.com/articulos/implementar-promesas-resolve-reject.html*

Mejoras en la programación orientada a objetos de ES6

Ahora vamos a abordar una de las mejoras más destacadas en ES6 y más demandadas por la comunidad de desarrolladores de Javascript. Se trata de una serie de nuevas características relacionadas con la Programación Orientada a Objetos.

Básicamente ahora Javascript con ES6 es capaz de implementar clases, algo que hasta ahora no estaba disponible en Javascript, donde solo teníamos objetos. Además, las clases en ECMAScript 2015 tienen comportamientos habituales en este paradigma de la programación como la herencia.

Clases en ES6

Qué son las clases y cómo se declaran en Javascript con la especificación ECMAScript 2015 (ES6).

En Javascript no existía una manera específica de crear clases, de programación orientada a objetos (POO). Sí que existían diversas alternativas que se podrían usar para crear componentes parecidos a lo que serían las clases en la POO tradicional, pero lo cierto es que no existía una declaración "class" como se podría esperar.

Así que en la versión reciente de Javascript ES6 una de las novedades más importantes fue la incorporación de la declaración de clases. Aunque las clases en Javascript siguen sin ser exactamente lo que puedas conocer en lenguajes más tradicionales como Java, sí resulta un avance determinante. Es importante que lo conozcas, ya que ha sido muy bien acogido por la comunidad y las mejores librerías y frameworks se han decidido a incorporar las declaraciones de clases como mecanismo de creación de sus diferentes componentes o artefactos.

En este artículo no pretendemos explicar el concepto de clase, algo que ya hemos tratado en diversas ocasiones anteriores, como en el Manual de Programación Orientada a Objetos, sino que nos queremos centrar más en la sintaxis y alternativas para definir clases en Javascript con ES6.

Declaración de una clase en Javascript ECMAScript 2015

En ECMAScript 2015 (ES6) las clases se declaran de manera similar a otros lenguajes, usando la palabra "class", seguida del nombre de la clase que estamos creando.

```
class Coordenada {

}
```

> **Nota:** Los nombres de las clases, igual que las variables o el lenguaje Javascript en general, son sensibles a mayúsculas y minúsculas. Podríamos llamarla como se desee, pero por convención se usa siempre la primera letra en mayúsculas.

Dentro de las llaves del "class" colocaremos el código de la clase. En lenguajes tradicionales serían típicamente los atributos y los métodos, pero en Javascript los atributos de instancia los tendremos que crear dentro del constructor.

> **Nota:** Otra posibilidad de declarar atributos en objetos es usar los getters de Javascript (ya disponibles en ES5), que permiten definir una especie de propiedad computada, cuyo valor se establece con la ejecución de un método. Si te interesa saber algo más te recomendamos un artículo donde puedes aprender más sobre los setters y getters de Javascript.

```
class Coordenada {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

Esta clase define un constructor, que se encarga de resumir las tareas de inicialización de los objetos. En la inicialización creamos los atributos que debe contener todo objeto que se cree a partir de esta clase.

> **Nota:** No puede haber más de un método llamado constructor en una clase de Javascript. Si por despiste colocamos más de uno recibiremos un mensaje de error "A class may only have one constructor". Esto indica que no es posible sobrecargar los constructores con las herramientas que te ofrece la creación de clases, aunque sí es posible hacerlo en la práctica, porque con Javascript puedes llamar a una misma función con distintos juegos de parámetros sin producir un error. El problema es que tendrías que implementar tú mismo mediante código la sobrecarga, recibiendo parámetros de manera predeterminada, examinando sus tipos, etc.

Instanciación de objetos a partir de una clase

Como en otros lenguajes, usamos la palabra "new" para crear nuevos objetos de una clase, seguida del

nombre de la clase y todos los parámetros que se deban indicar para invocar al constructor.

```
var testCoordenada = new Coordenada(8,55);
```

El proceso de instanciación creará el nuevo objeto y su inicialización quedará en manos del constructor. En nuestro caso el construtor había creado dos propiedades para cada instancia, a los que les asignaba los valores de los puntos de la coordenada.

Los objetos poseen propiedades, también llamados atributos, que se hayan definido en el momento de creación de la clase o en cualquier otro momento de la vida de ese objeto. En nuestro caso se crearon los atributos "x" e "y".

> **Nota:** La palabra "this" dentro del constructor, o cualquier otro método de la clase, hace referencia al objeto que ha recibido ese método. Si es el constructor hace referencia al objeto que se está creando. Si es en un método hace referencia al objeto que recibió el mensaje (invocación del método).

Una vez creados los objetos podríamos acceder a sus atributos a partir del operador punto ("."). Usamos el nombre del objeto que acabamos de crear, el operador punto y el nombre del atributo al que queramos acceder.

```
console.log(testCoordenada.x);
```

Eso nos imprimiría en la consola el valor del atributo "x" del objeto que se había creado anteriormente.

Creación de métodos en la clase

Las clases en ES6 pueden declarar sus métodos de una manera resumida, pues nos ahorramos la palabra "function" en la declaración de la función.

> **Nota:** Un método es una función asociada a un objeto. Por ejemplo, la clase "puerta" puede tener el método "abrir" y entonces todos los objetos puerta podrán recibir el mensaje "abrir", provocando la apertura de la puerta. Los métodos los puedes entender como funciones, solo que se ejecutan en el contexto de un objeto que haya sido creado.

En el código del ejemplo de la coordenada ya teníamos un método, aunque algo especial, ya que era el método constructor. El constructor debe tener siempre el nombre "constructor", pero podríamos crear otros métodos con cualquier otro nombre.

```
class Coordenada {
  constructor(x, y) {
    this.x = x;
```

```
    this.y = y;
  }

  esIgual(coordenada) {
    if (this.x == coordenada.x && this.y == coordenada.y) {
      return true;
    }
    return false;
  }
}
```

En el código anterior hemos agregado el método "esIgual" que se encarga de decir si una coordenada recibida por parámetro tiene los mismos valores de x e y que la coordenada que ha recibido el mensaje.

> **Nota:** La definición de métodos sin la palabra "function" es una de las novedades interesantes en ES6, azúcar sintáctico, disponible también a la hora de crear objetos a partir de literales.

Recuerda que los métodos se invocan sobre el nombre del objeto, con el operador punto y el nombre del método, seguido por sus parámetros. Podríamos usar esta clase y el nuevo método con este código:

```
var testCoordenada = new Coordenada(8,55);
console.log(testCoordenada.esIgual(new Coordenada(3, 1)))
```

Al ejecutar esto, veríamos "false" en la consola de Javascript.

Declaraciones de clases siempre antes de su uso

Es indispensable que se declaren las clases antes de usarlas. A diferencia de las funciones en Javascript, que se pueden usar antes de haber sido declaradas, las clases deben conocerse antes de poder instanciar objetos.

Es decir, un código como este no funcionaría:

```
var x = new MiClase();
class MiClase {}:
```

Si intentas ejecutarlo, recibirías por consola un mensaje como este: "Uncaught ReferenceError: MiClase is not defined".

Expresiones de clases ES6

Otra manera de declarar clases en ES6 es mediante lo que se conoce como expresiones. Básicamente consiste en crear una variable y asignarle una expresión definida mediante class y las llaves.

```
var Persona = class {
```

```
  constructor(nombre) {
    this.nombre = nombre;
  }
}
```

La clase puede ser anónima, como en el ejemplo anterior, aunque también podría tener un nombre.

> **Nota:** Esto es igual que las funciones, que puedes asignarle una función a una variable y es similar a haber declarado la función como se hace tradicionalmente. Y digo similar, porque al apoyarte en una variable en realidad sí hay diferencias, pues el ámbito de la variable que estás definiendo (var Persona) puede ser diferente al ámbito de la clase si la hubieras declarado de manera tradicional, debido al hoisting de Javascript. A mi juicio no aporta mucho este modo de declaración de clases y no lo he visto usar en ninguna documentación.

El uso de una clase definida con una expresión, a la hora de instanciar objetos, es exactamente el mismo que hemos visto anteriormente.

```
var miguel = new Persona('Miguel Angel Alvarez');
console.log(miguel.nombre);
```

Ahora ya sabes declarar clases en ES6, así como definir métodos. Has aprendido a usarlas para instanciar objetos y luego a usar esos objetos para el acceso a sus miembros, tanto sus atributos como sus métodos.

En siguientes artículos exploraremos más características de ECMAScript 2015 relacionadas con las clases, como la herencia en las clases de ES6.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *25/07/2017*
Disponible online en *http://desarrolloweb.com/articulos/clases-en-es6.html*

Uso de static en las clases Javascript ES6

Con la palabra static podemos definir métodos estáticos en Javascript, versión ES6. Explicamos su funcionamiento.

En este artículo vamos a conocer cómo trabajar con miembros de clase en las clases de ES6, la versión reciente del estándar de Javascript. Aprenderemos a crear métodos static y propiedades también, aunque verás que para el caso de las propiedades tendremos que valernos de algún truquillo adicional.

En realidad no es nuestro objetivo introducir el concepto de static, pues es motivo de estudio en detalle en artículos anteriores como Explicaciones de métodos y atributos static. No obstante diremos que los miembros estáticos de las clases en Programación Orientada a Objetos en general son atributos y métodos que dependen directamente de una clase, en vez de depender de un objeto en particular. Al depender de la

clase no están asociados a un objeto, por lo que comparten el valor para toda la clase, independientemente de las instancias que se hayan creado.

Ya hemos comentado que con la reciente versión de Javascript ES6 ahora disponemos de clases, aunque no son exactamente lo mismo que en otros lenguajes más tradicionales. En el caso de static no hay cambios en el concepto, pero al no poderse declarar atributos de instancia, tampoco podremos declarar atributos static. Lo veremos también con calma, aunque antes comencemos con los métodos estáticos.

Definir métodos static en ES6

Un método estático se construye simplemente indicando la palabra "static" antes del nombre del método que se está creando. El resto de la definición de un método estático sería igual que la definición de un método convencional, con la excepción de disponer de la variable "this" como habitualmente en los métodos.

En el siguiente ejemplo tenemos una clase llamada "Sumatorio" que tiene un método declarado estático, para sumar los valores de un array.

```
class Sumatorio {
  static sumarArray(arrayValores) {
    let suma = 0;
    for(let valor of arrayValores){
      suma += valor
    }
    return suma;
  }
}
```

El método static depende directamente de la clase, por lo que usaremos la propia clase para invocarlo.

```
let suma = Sumatorio.sumarArray([3,4,5]); //suma valdrá 12
```

Nota: El hecho de no poder disponer de "this" dentro de un método estático es debido a que el método no se invoca con relación a ningún objeto. Como has visto, usamos el nombre de la clase para invocarlo y no un objeto instanciado. Como sabes, "this" tiene una referencia al objeto donde se lanzó un mensaje (el objeto sobre el que se invocó un método). Como no existe tal objeto de invocación, no existe un objeto en "this". En principio podríamos pensar que "this" entonces valdrá "undefined", pero lo que hay en realidad es el código de la propia clase.

Los métodos estáticos pueden servir para muchas cosas. Es el motivo por lo que a veces se usan como un cajón desastre de utilidades que puedan tener que ver con una clase. Pensando en objetos hay que tener cuidado para qué y cómo se utilizan. El pasado ejemplo de sumarArray() no era muy bueno desde la filosofía de la orientación a objetos, pero en el siguiente ejemplo tenemos un método estático un poco mejor pensado.

Tenemos una clase Fecha que nos sirve para crear fechas en Javascript. Es cierto que Javascript contiene ya una clase Date, pero tiene la posibilidad de crear fechas y horas y quizás nosotros solo necesitamos fechas y queremos una serie de utilidades adicionales que no están incluidas en la interfaz original de Date.

En nuestro ejemplo observarás que tenemos un constructor, que recibe el día, mes y año. Sin embargo en la práctica muchas veces las fechas se crean con el día actual. Como no existe la sobrecarga de métodos en Javascript y por tanto tampoco podemos sobrecargar constructores, podríamos echar mano de los métodos static para crear una especie de constructor de la fecha sin parámetros que nos devuelve un objeto Fecha inicializado con el día actual.

```
class Fecha {
  constructor(dia, mes, ano) {
    this.dia = dia;
    this.mes = mes;
    this.ano = ano;
  }

  static hoy() {
    var fecha = new Date();
    var dia = fecha.getDate();
    var mes = fecha.getMonth() + 1;
    var ano = fecha.getFullYear();
    return new Fecha(dia, mes, ano);
  }
}
```

Como puedes ver, el método static hoy() se encarga de obtener los valores del día, mes y año actuales e invocar al constructor con tales datos, devolviendo el objeto que se acaba de crear.

Otro ejemplo de método estático o método de clase

Continuamos con un segundo ejemplo de método estático o método de clase. Ahora lo encontramos en el marco de una clase Coordenada,

```
class Coordenada {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static coordenadaOrigen() {
```

```
    return new Coordenada(0,0);
  }
}
```

En el código anterior tienes un ejemplo de método estático, llamado coordenadaOrigen(), que devuelve una nueva instancia de un objeto de la clase Coordenada, con sus putos x e y igual a cero.

Ese es un método de clase, por lo que tendremos que usar la propia clase para acceder a él.

```
var origen = Coordenada.coordenadaOrigen();
```

Atributos estáticos de ECMAScript 2015

La definición de propiedades estáticas, o propiedades de clase, no es tan directa como la definición de métodos estáticos, puesto que en ES6 no se pueden definir propiedades tal como se hace en otros lenguajes de programación más tradicionales.

En ECMAScript 2015 (ES6) tenemos la limitación de no poder declarar atributos en la clase (tenemos que generarlos en el constructor o en los métodos). Esto también se extiende a los atributos de clase o atributos static. Sin embargo, siempre te puedes montar tú mismo algún mecanismo para conseguirlo.

Por ejemplo en el caso de tener atributos de clase estáticos que tengan valores comunes a toda la clase, podríamos hacer uso de los getter, colocando la palabra static a la hora de definir el método get.

```
class Circulo {
  static get pi() {
    return 3.1416
  }
}
```

Podremos acceder a "pi" como si fuera una propiedad estática, dependiente directamente de la clase. La usamos directamente desde el nombre de la clase:

```
console.log(Circulo.pi);
```

Si lo que queremos es una variable estática, que sea global para toda la clase, con un valor que no depende de las instancias y que puede variar a lo largo del tiempo, podríamos hacer algo como esto:

```
class Test {
}
Test.variableStatic = 'Algo que guardo en la clase';
```

Como Javascript es tan permisivo, podemos asociar una propiedad a la clase simplemente asignando cualquier valor. No me gusta demasiado el ejemplo, porque la definición de la propiedad estática estaría

fuera del código de la propia clase y por tanto en una lectura a ese código podríamos no darnos cuenta que más adelante se crea esa variable estática.

En el ejemplo típico de crear una variable estática que lleva la cuenta de las instancias creadas a partir de una clase, podríamos optar por algo como esto (que me gusta más por tener la creación de la propiedad estática dentro del constructor).

```
class Habitante {
  constructor(nombre) {
    this.nombre = nombre;
    if(Habitante.contador) {
      Habitante.contador++;
    } else {
      Habitante.contador = 1;
    }
  }
}
```

El problema aquí es que únicamente existirá esa propiedad estática a partir de la primera instanciación de un objeto. Así que otro ejemplo un poco más enrevesado podría ser el siguiente, que hace uso de los getter y de los setter de los objetos Javascript.

```
class Habitante {

  static get contador() {
    if(Habitante.contadorPrivado) {
      return Habitante.contadorPrivado;
    }
    return 0;
  }

  static set contador(valor) {
    Habitante.contadorPrivado = valor;
  }

  constructor(nombre) {
    this.nombre = nombre;
    if(Habitante.contador) {
      Habitante.contador++;
    } else {
      Habitante.contador = 1;
    }
  }

}
```

Como puedes ver, Habitante.contador es nuestra propiedad estática, que estaría disponible gracias a los getter y los setter como si fuera un atributo normal (solo que es estático por estar predecido de "static").

En el constructor hacemos uso de Habitante.contador como si fuera un atributo normal, solo que

internamente en la implementación de la clase estas propiedades en realidad se calculan con funciones computadas get y set.

Es un código meramente experimental, pero te puede dar una idea de las cosas que se pueden hacer en Javascript cuando "retuerces" un poco el lenguaje. Aprende más sobre los get y los set en este artículo.

Conclusión sobre los miembros de clase en ES6

Hemos aprendido cosas interesantes sobre la creación de miembros de clase, o miembros estáticos, en las clases de ES6. Como has podido ver, existen algunas particularidades dadas por el lenguaje Javascript, que es importante conocer.

En el siguiente artículo del Manual de ES6 seguiremos hablando de clases, abordando algo tan importante como la herencia de clases en Javascript.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *30/11/2017*
Disponible online en *http://desarrolloweb.com/articulos/static-clases-javascript-es6.html*

Herencia en clases con Javascript ECMAScript 2015

Explicamos los mecanismos y la sintaxis de herencia en las clases de Javascript, disponibles a partir de ES6 (ECMAScript 2015).

En el artículo anterior conocimos las clases en ES6 con detalle, creación de clases, métodos, constructores, etc. No obstante, nos quedan cosas por aprender y ahora nos vamos a dedicar a conocer la herencia de la Programación Orientada a Objetos. Estudiaremos brevemente lo que consiste la herencia, pero sobretodo nos centraremos en la manera de implementarla en el lenguaje Javascript a partir de la especificación ECMAScript, edición de 2015.

A nivel conceptual, la herencia es un mecanismo por el cual unas clases pueden ser construidas en base a otras. Las clases originales, que nos sirven de base, podemos llamarlas padres y a las clases que se construyen en base a los padres podemos llamarlas hijos. Ante la herencia, las clases padre transmiten sus miembros (atributos y métodos) a las clases hijo. También se dice que la clase hija deriva de la clase padre.

Nota: No vamos a agregar mucho más de teoría, pues entendemos que el lector la conoce. Si no es así, recomendamos la lectura del artículo sobre Herencia en la Programación Orientada a Objetos.

Extender clases en ES6

Extender una clase es el mecanismo por el cual una clase se construye en base a otra, es decir, el mecanismo mediante el cual se construyen clases hijas o clases derivadas. Obviamente, para extender, lo primero que debemos tener es la clase padre y mediante esta extensión construir una clase hija.

En Javascript, igual que en muchos otros lenguajes, para extender una clase en base a otra usamos la palabra "extends".

```
class Coordenada3D extends Coordenada {

}
```

En este caso, la clase Coordenada3D es la clase hija y la clase Coordenada es la clase padre. Por tanto, la clase Coordenada3D hereda todas las propiedades y métodos existentes en la clase Coordenada.

> **Nota:** Tienes el código de la clase Coordenada en el artículo anterior sobre las clases en ES6.

En el caso particular de las clases de Javascript vimos que las propiedades se crean en tiempo de ejecución, es decir, no se declaran, por lo que en realidad lo que estamos heredando son únicamente los métodos.

Invocación al constructor de la clase padre

Es algo habitual que la clases hijas se apoyen en los constructores de las clases padre para poder hacer sus tareas de inicialización y creación de las propiedades o atributos de la clase. Para ello es posible invocar al método constructor de la clase padre, dentro del código del constructor de la clase hija.

El mecanismo para invocar a métodos existentes en la implementación del padre es mediante la palabra "super" y los paréntesis de invocación de métodos.

```
class Coordenada3D extends Coordenada {
  constructor(x, y, z) {
    super(x, y);
    this.z = z;
  }
}
```

Una coordenada de 3 dimensiones se define mediante tres puntos (x, y, z). Así pues, el constructor de la Coordenada3D debe inicializar esos tres puntos. La clase Coordenada original (clase padre) ya inicializaba dos de ellos, por lo que no necesitamos repetir el código que había en Coordenada para esas inicializaciones. En el código anterior podemos apreciar cómo para la inicialización de los atributos "x" e "y" se invoca al constructor de la clase padre, mediante super(x, y). Ya solo nos quedaría inicializar el tercer atributo "z".

> **Nota:** Al extender una clase tiene todo el sentido apoyarse en el código de la clase hija. En este caso en concreto no se ahorra demasiado código, pero en clases mayores sin duda el ahorro de código es mucho mayor. Además, en la mayoría de los casos el beneficio no es solamente evitar repetir unas pocas líneas de código, sino evitar mantener el mismo código en dos lugares distintos.

Obligación de invocar a super() en el constructor de las clases derivadas

Hasta aquí todo te sonará de otros lenguajes, si es que ya tienes nociones de programación orientada a objetos. Sin embargo en Javascript hay un detalle importante que llama la atención en el caso de los constructores de clases derivadas. Básicamente resulta que, si quieres usar "this" en el constructor de la clase hija, estás obligado a llamar a super() previamente.

Es decir, como en la práctica siempre vas a querer usar this para referirte al objeto que se está construyendo, necesitarás invocar a super() siempre con antelación, para que el constructor de la clase padre haga su trabajo, antes de comenzar a operar con el objeto que se está construyendo en el contexto de la clase hija.

Por ejemplo, el siguiente código produciría un error: "Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor"

```
class Coordenada3D extends Coordenada {
  constructor(x, y, z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```

Tampoco podrás hacer esto:

```
class Coordenada3D extends Coordenada {
  constructor(x, y, z) {
    this.z = z;
    super(x, y);
  }
}
```

Sobreescritura de métodos

La redefinición o sobreescritura de métodos es una de las tareas habituales al extender clases. Consiste en reprogramar ciertas operaciones en la clase hija, algo que se traduce en la reescritura del código de los métodos que teníamos en el padre. Para conseguirlo simplemente se tiene que volver a declarar el método con el mismo nombre existente en la clase padre.

La reescritura del constructor, explicada en el paso anterior, es un caso particular de la redefinición de métodos. En general, no solamente tendremos que programar un nuevo constructor. Realmente, al extender la clase hija es habitual que tengamos que redefinir métodos existentes en la clase padre.

Como en el caso del constructor, podemos invocar al método original, tal como se había implementado en la clase padre, mediante super().

```
class Coordenada3D extends Coordenada {
  constructor(x, y, z) {
    super(x, y);
    this.z = z;
  }

  esIgual(coordenada3D) {
    if(super.esIgual(coordenada3D) && this.z == coordenada3D.z) {
      return true;
    }
    return false;
  }
}
```

En este caso hemos realizado la redefinición del método esIgual(), en la que se tiene que controlar si la coordenada tiene iguales los tres puntos. Dos de ellos se verifican invocando al método de la clase padre y el otro es tarea de la clase hijo.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *27/07/2017*
Disponible online en *http://desarrolloweb.com/articulos/herencia-clases-javascript-ecmascript.html*