Los nuevos operadores, métodos y sintaxis de ECMAScript están orientados a ofrecer un código más claro, legible y menos extenso. Al principio es complejo de entender, puesto que el código varía de manera considerable entre una versión y otra. Aunque os acostumbraréis pronto al ver como el código de vuestros proyectos se reduce y el resultado es más limpio y manejable.

A continuación vamos a ver los operadores, métodos y sintaxis más utilizados de las nuevas versiones de ECMAScript. Os recomiendo escribir los ejemplos y vuestras primeras líneas de código en la consola interactiva de BabelJS o REPL. De esta manera comprenderéis mejor cómo funciona el código y el resultado de la compilación.

Const, let

La principal diferencia de `const` y `let` con `var`, es que el contexto (scope) de estos tipos de se restringen al del bloque en vez de al de la función.

```
var myFunction = function() {
  for(var a = 1; a < 3; a++) {}
  for(let b = 1; b < 3; b++) {}

  console.log(a); // 3
  console.log(b); // b no está definida!
}
```

Otro ejemplo:

```
var myFunction = function() {
  var a = 1;
  let b = 1;

  if (true) {
    var ab = 3;
    let bb = 3;
    // Asignamos nuevos valores
    a = 2;
    b = 2;
  }

  console.log(a); // 2
  console.log(b); // 2
  console.log(ab); // 3
  console.log(bb); // bb no está definida!
}
```

Un último ejemplo:

```
var myFunction = function() {
  var a = 1;
  let b = 1;

  if (true) {
    var a = 3;
    let b = 3;
  }

  console.log(a); // 3
  console.log(b); // 1
}
```

En resumen, podríamos decir que `let` es el nuevo `var`. `const` se comporta igual que `let` exceptuando que no se puede modificar su valor una vez asignado. Como su propio nombre indica `const` es una constante.

Código en Babel REPL.

Arrow functions (=>)

Propuesta: Arrow Functions.

Este operador nos permite escribir funciones de una manera más compacta. No necesitamos utilizar la palabra clave `function`, sino que asignamos la función directamente a una variable:

```
const sum = (a, b) => { return a + b };
```

`(a, b)` representa los argumentos de la función. Para un solo argumento los paréntesis son opcionales:

```
const multiplyBy2 = a => { return 2 * a };
```

Si nuestra función tiene solo una línea de código y esta corresponde a `return`, podemos eliminar las llaves y la palabra clave `return`:

```
const multiplyBy2 = a => 2 * a;
```

Esta sintaxis es muy útil al combinarla con métodos de iteración:

```
let myArray = [1, 2, 3, 4],
  even = myArray.filter(n => n % 2 === 0);
```

Lo más importante de este operador es que asigna el valor de la variable `this` al contexto actual. Esto quiere decir que `this` se restringe al contexto, a partir de ahora *scope*, en el que se ha definido la función. Esta propiedad nos será muy útil a la hora de trabajar con clases. Podemos ver la diferencia del valor de `this` en el siguiente ejemplo: Context y operador =>.

Si queréis investigar más sobre ECMAScript y contextos, os recomiendo este artículo de Ryan Morr.

Template strings

Esta funcionalidad simplifica la tarea de generar cadenas de texto (*strings*). Con esta sintaxis podemos insertar variables y escribir cadenas de texto de varias líneas sin recurrir a la concatenación con el operador `+`:

```
// String literal
let str = `Esta es una string.`
// Multilinea
let multiline = `
Esta es una string
multilinea`;
// Interpolación
let name = 'Mario';
let interpolation = `Hello ${name}`;
```

Default parameters

Una funcionalidad muy común en otros lenguajes es la posibilidad de establecer parámetros en funciones con un valor defecto. En el caso en el que no se especifique el valor del parámetro al llamar a la función, este tomará el que se ha definido por defecto.

```
const multiply = (a = 5, b = 1) => { console.log(a * b); }

multiply() // 5
multiply(2) // 2
multiply(3, 2) // 6
```

También podemos utilizar los parámetros por defecto con la sintaxis de `function`:

```
const sum = function(a = 5, b = 0) {
  console.log(a + b);
}

sum() // 5
sum(2) // 2
sum(3, 2) // 5
```

Código en Babel REPL.

Shorthand properties

En *ES5* utilizamos los `:` para definir las propiedades de un objeto:

```
let a = 1;
let b = 2;
let obj = { a: a, b: b };
```

Si analizamos este caso vemos que la información es redundante, puesto que el nombre de la clave del objeto es el mismo que el de la variable que le da valor.
En *ES2015*, si el nombre de la clave coincide con el de la variable se puede omitir:

```
let a = 1;
let b = 2;
let obj = { a, b };
```

Código en Babel REPL.

Spread/Rest operator (...)

Propuesta: Object Rest/Spread Properties for ECMAScript.

El operador `...` se utiliza para desestructurar objetos. Partiendo de un objeto base podemos extraer sus claves y asignarlas a una variable:

```
let base = { a: 1, b: 2, c: 3 };
// Si lo asignamos sin el operador (...). En este ejemplo hacemos uso
// de las shorthand properties
let noSpread = { base }; // noSpread = { base: { a: 1, b: 2, c: 3 } }
```

```
let noSpread = { base }; // noSpread = { base: { a: 1, b: 2, c: 3 } }
// En el caso de utilizar spread
let withSpread = { ...base } // withSpread = { a: 1, b: 2, c: 3 }
```

Podemos combinarlo con otras propiedades en el objeto:

```
let base = { a: 1, b: 2, c: 3 };
let d = 4;

let withSpread = { ...base, d } // withSpread = { a: 1, b: 2, c: 3, d: 4 }
let withSpread2 = { d, ...base } // withSpread = { d: 4, a: 1, b: 2, c: 3 }
```

Este operador también se utiliza para extraer las propiedades restantes de un objeto y asignarlas a distintas variables:

```
let base = { a: 1, b: 2, c: 3, d: 4 };
// Asignamos los valores
let { a, b, ...other } = base; // a = 1, b = 2, other = { c: 3, d: 4 }
```

Código en Babel REPL.

Object.assign

Este método nos permite crear un nuevo objeto partiendo de otros. Para ello generar el nuevo objeto se combinan las propiedades de todos los argumentos del método:

```
let base = { name: 'Assign!' };
let obj1 = { a: 1, b: 2 };
let obj2 = { c: 3 };
let obj3 = { d: 4, e: 5 };

let newObj = Object.assign({ prop: 'test'}, base, obj1);
// {  a: 1, b: 2, name: 'Assign!', prop: 'test' }
```

Código en Babel REPL.

Cómo podéis ver en el ejemplo, este método es el que utiliza Babel al compilar código que incluye el operador `...` .

Async / await

Propuesta: Async Functions for ECMAScript

Las promesas de *ES2015* han sido un gran paso a la hora de trabajar con código asíncrono en ECMAScript. El problema de esta solución es la cantidad de código y funciones que hay que introducir para transformar un código síncrono en asíncrono. Para solventar este problema, *ES2017* incluirá los nuevos operadores `async` / `await` .

Para explicar estas nuevas funciones, vamos a partir de un código síncrono:

```
function slowMethod() {
  let result = getResult(), finalResult;
  // Realizamos operaciones sobre result...
  return finalResult;
}

// En otra parte de nuestro código
result = slowMethod();
```

Supongamos que `getResult` realiza tarda cierto tiempo en obtener el resultado. Cada vez que ejecutemos `slowMethod` tendremos que esperar a obtener el resultado, bloqueando al resto de la aplicación. Con *ES2015* podemos reescribir este código utilizando las promesas:

```
function slowMethod() {
  return getResult().then(result) {
    let finalResult;
    // Realizamos operaciones sobre result...

    return finalResult;
  }
}

// En otra parte de nuestro código
slowMethod().then(finalResult) { /* ... */ };
```

Si comparamos el código síncrono y asíncrono estos varían considerablemente. Con las promesas necesitamos concatenar el método `then` en cada llamada. En cambio, si utilizamos `async` / `await` podemos convertir nuestro código en asíncrono de manera muy sencilla:

```
async function slowMethod() {
  let result = await getResult(), finalResult;
  // Realizamos operaciones sobre result...
  return finalResult;
}

// En otra parte de nuestro código
result = await slowMethod();
```

Utilizando estas nuevas funciones el código queda mucho más claro y similar al código síncrono.

Y aún hay mucho más

Estos son las nuevas funcionalidades de ECMAScript más útiles a la hora de trabajar con React. Os las encontraréis en multitud de ocasiones. Si queréis conocer todas las funcionalidades que están por venir, no dudéis en indagar en la sección ES2015 and beyond de la web de Babel.