NOVEDADES ES6

Arrow Functions

Conocidas en otros lenguajes (C#, Java) como expresiones lambda, arrows o flechas son abreviaciones de funciones utilizando el operador =>

Por ejemplo, cuando queremos asignar un callback a una función, debemos escribir la función completa, supongamos que queremos obtener los cuadrados de los números contenidos en un arreglo:

```
var cuadrados = numeros.map(function(n) {
  return n * n;
});
```

Con la nueva sintaxis podemos obtener el mismo resultado pero de una forma más elegante:

```
var cuadrados = numeros.map(n => n * n);
```

Clases

Si tienes experiencia con otros lenguajes orientados a objetos como Java te habrás dado cuenta que las clases en JavaScript se creaban de una forma diferente, ahora la sintaxis ha cambiado para hacerla más parecida a lo que estamos acostumbrados, en este ejemplo definimos una clase con dos atributos y dos métodos:

```
class Rectangulo {
  constructor(base, altura) {
    this.base = base;
    this.altura = altura;
  }
  calcArea() {
    return this.base * this.altura;
  }
}
var r = new Rectangulo(5, 10);
console.log(r.calcArea()); // 50
```

Template Strings

Son un tipo especial de cadena con formato, similares a la interpolación en otros lenguajes como Ruby, se definen con un par de caracteres back-tick (`) a diferencia de las cadenas normales que usan comillas sencillas o dobles.

```
var s1= 'esta es una template string';

// Pueden contener valores
var n = 5;
var s2 = `El valor de n es ${n}`;

// Pueden abarcar múltiples líneas
var s3 = `Esta es una cadena
escrita en dos líneas`;
```

Let y Const

let indica que una variable solo va a estar definida en un bloque en particular, al terminar el bloque la variable deja de existir, esto es muy útil para evitar errores lógicos cuando alteramos una variable que no deberíamos.

```
function letTest() {
  if (true) {
    let x = 23;
    console.log(x);  // 71
  }
  console.log(x);  // no existe x
}
```

Generadores

Los generadores son un tipo especial de función que retornan una serie de valores con un algoritmo definido por el usuario, podríamos generar ids consecutivos para registros de una base de datos, sucesiones numéricas como Fibonacci, entre otras. Una función se convierte en generador si contiene una o más expresiones yield y se declara con function*.

En este ejemplo definimos un generador que me devuelva el cuadrado de los números empezando con 1, es decir que los primeros cinco valores serían 1, 4, 9, 16, 25:

```
function* cuadrados(){
  var n = 1; // comienza en 1
  while(true) {
    var c = n * n; // obtiene el cuadrado
    n++; // aumenta para la próxima iteración
    yield c; // devuelve el valor actual
  }
}
```

Literales octales y binarias

Hay ocasiones en que el contexto de nuestros datos requiere que trabajemos con cifras no decimales, por ejemplo, en base 2 (binario) o base 8 (octal), ahora es sencillo crear este tipo de literales con los prefijos (0b) y (0o) respectivamente.

```
var a = 0b111110111; // binario
console.log(a); // 503
var b = 0o767; // octal
console.log(b); // 503
```

Maps y Sets

A la hora de desarrollar algoritmos en JavaScript solo teníamos dos alternativas crear un objeto {} o un arreglo [] para almacenar datos, y aunque la mayoría de las veces es más que suficiente con ellos, puede presentarse el caso de que el rendimiento no es el adecuado. Los mapas (Map) son una estructura de datos que almacenan pares de llave (key) y valor (value), los conjuntos (Set) tienen la característica de no aceptar duplicados, y ambos permiten búsquedas eficientes cuando se tiene un gran volumen de información porque no guardan sus elementos ordenados por un índice, como ocurre con los arreglos.

En el siguiente ejemplo vemos el uso de Set, noten como se puede encadenar el método add para añadir nuevos elementos, si alguno se duplica es omitido y tiene un método has para revisar si existe un elemento dentro del conjunto.

```
// Sets
var s = new Set();
// Añade 3 elementos, cadena1 se repite
s.add("cadena1").add("cadena2").add("cadena1");
// El tamaño es 2
console.log(s.size === 2);
// El conjunto no tiene la cadena hola
console.log(s.has("hola"));
```

Promises

El flujo de información de Internet tiene características asíncronas, lo que significa que mientras esperamos el resultado de una operación como por ejemplo que carguen los datos de una página web, un programa puede realizar otras operaciones y cuando el resultado esté listo utilizarlo. Las promesas son objetos que representan esta clase de operaciones y los datos que se obtienen.

En el siguiente bloque de código el método obtenerDatos hace una petición a algún otro sitio y devuelve un dato de interés para nosotros, el problema es que ese proceso puede demorar algunos milisegundos y nosotros no podemos esperar, entonces en vez de devolvernos el dato como tal, nos devuelve un objeto de tipo Promise que tiene un método then, este método recibe dos funciones una que se ejecuta en caso de que se obtenga el dato correctamente (success) y la otra en caso de que haya ocurrido un error (failure), una de estas funciones se ejecutará en el futuro cercano.

```
var promesa = obtenerDatos();
promesa.then(function (dato) {
  console.log(dato);    // 'mensaje'
}, function (error) {
  console.error(error);   // ocurrió un error
});
```

Cada variable tiene un nombre y cumple una función específica, la forma de declaración de cada una es similar. Pero lo que cambia es la función que realiza esta.

Hoisting: Lo que yo he entendido es que podemos llamar a cualquier variable, aunque se nombre abajo, ya que cuando ejecutemos el programa, JS lo ejecutara la primera.