{desafío} latam_

# Introducción a ES6+ (Parte II)

## Elementos básicos de ES6+

### Competencias

- Identificar el concepto de programación funcional y orientación a eventos.
- Identificar el significado de asincronía y procesos paralelos, tanto como en cliente y en servidor.
- Interpretar los beneficios que implica trabajar de forma asíncrona.
- Ejecutar un programa JavaScript utilizando las características nuevas de variables, strings y funciones de JavaScript ES6+.

### Introducción

Luego de aprender a configurar un entorno, ahora nos toca conocer ES6+ en detalle. En este sentido. abordaremos algunos detalles interesantes acerca de la sintaxis de ES6+, la nueva definición de variables y sus diferencias. Por otra parte, veremos las nuevas funciones aplicadas a strings y objetos.

Todas las funcionalidades implementadas para este lenguaje no son azarosas, están pensadas para transformar JavaScript a un concepto de un nuevo paradigma de programación, actualmente dicho paradigma es clave para el desarrollo de tecnología asíncrona, procesos paralelos, elementos que serán explicados de igual forma en esta unidad. Es importante destacar, que el factor clave para entender este concepto es la recursividad y la posibilidad de poder invocar una función sobre la misma. El llevar este concepto a fondo se conoce como programación funcional. Dicho paradigma será abordado a continuación.

### ¿Qué es el paradigma funcional?

Es necesario señalar que el concepto de programación, como tal, nace como una rama de la matemática. Los elementos que normalmente nos toca programar están pensados principalmente en resolver algoritmos o problemas en donde la raíz de estos son matemática. Cuando damos solución a un problema, independiente del que sea, la matemática, directa o indirectamente está ahí.

Los primeros lenguajes de programación fueron diseñados para resolver complejos cálculos matemáticos que el ser humano, por cuenta propia, le era imposible. Al finalizar la segunda guerra mundial, luego de que la máquina de turing se diera a conocer, muchos investigadores comenzaron a desarrollar diversas formas de abordar problemas. En este sentido, los paradigmas de programación que se volvieron más populares fueron los imperativos y declarativos, por otra parte, otros investigadores crearon un concepto distinto, manteniendo la línea científica y matemática de la programación. Así es como nace la programación funcional.

En la programación funcional el principal protagonista es la función, en contraste con los lenguajes imperativos, declarativos y secuenciales en donde las variables, los ciclos y los condicionales eran lo más importante, en la programación funcional lo que destaca es la función y eso es principalmente por la recursividad.

## Entendamos la recursividad

La recursividad es una característica de algunos procesos u objetos que hacen referencia a sí mismos en su definición.

¿Qué queremos decir con esto? Primero, que la recursividad o recursión (las dos son correctas) no son una “cosa”, sino una “característica” de una cosa. Existen series recursivas, algoritmos recursivos, funciones recursivas… Así mismo, no hay una sintaxis especial o palabra clave para indicar que una función es recursiva, si no que simplemente es una forma de describir funciones que tienen la característica de que como parte del cuerpo de la función, encontramos una o más invocaciones a sí misma.

Como programadores, siempre estamos tratando de encontrar abstracciones que nos permitan evitar la repetición explicita. Por eso, cuando vemos en nuestro código que algo se repite varias veces, siempre nos planteamos refactorizar para “abstraer” ese pedacito que se repite en una variable, una función, clase, bucle, para poder reusarlo.

La recursión nos ofrece una forma de pensar y organizar procesos repetitivos, al igual que los bucles. De hecho, todo algoritmo recursivo puede implementarse usando iteración. La recursión, a través del uso de funciones, nos permite abstraer y representar ciclos repetitivos de una forma alternativa a la iteración. Por ejemplo:

```
const imprimeElementos = (arr) => {
  for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
  }
}
imprimeElementos([1, 2, 3, 4]);
```

El código de arriba recibe un array e imprime todos sus valores en la consola, uno a uno, usando un bucle for. La misma lógica se puede expresar de forma recursiva:

```
const imprimeElementos = (arr) => {
  if (arr.length > 0) {
    console.log(arr.shift());
    return imprimeElementos(arr);
  }
};
imprimeElementos([1, 2, 3, 4]);
```

La nueva versión de `imprimeElementos()` produce el mismo resultado, pero en vez de usar un bucle, usa recursión para recorrer el arreglo.

Básicamente lo que hace la función es imprimir el primer elemento del arreglo, al aplicar la función `.shift()` lo que se logra es que al imprimir el elemento se quita este del arreglo quedando en cada iteración el arreglo de la siguiente forma:

```
[1, 2, 3 ,4] //iteración 0
[2, 3 ,4] //iteración 1
[3 ,4] //iteración 2
[4] //iteración 3
[] //iteración 4
```

Como se valida con una condicional que el arreglo tenga al menos un elemento, al no tener elementos se cierra el ciclo generado por la recursividad y se da por finalizada la función.

Ahora que ya hemos visto, en líneas generales, el significado de la recursión, veamos algunos ejemplos un poco más reales. De hecho, como desarrolladores web, hay una estructura en particular con la que nos encontramos a diario, donde la recursión va a ser una herramienta esencial: el DOM.

Para “recorrer” este tipo de estructura, donde no sabemos la “profundidad” (los niveles de anidación), y donde cada elemento es “similar” (con las mismas características), una función recursiva es la ruta más obvia. Imaginemos que tenemos que implementar una función que dada una condición, a comprobar para cada nodo, retorna un arreglo con todos los nodos que cumplan la condición:

```
const select = (element, condition) => {
  let found = [];
  if (condition(element)) {
    found.push(element);
  }
  for (let i = 0; i < element.children.length; i++) {
    found = found.concat(select(element.children[i], condition));
  }
  return found;
};
```

Una vez implementada nuestra función `select()` , podríamos invocarla así:

```
const selection = select(
  document.body,
  element => element.tagName === 'H2'
);
```

No te preocupes si la notación de estos ejemplos te parece un poco compleja de leer, lo importante de dichos ejemplos es el concepto, como es imposible determinar la profundidad de un nodo sin conocerlo previamente, entonces se construye una función recursiva que permita llegar a los hijos.

La recursividad se ocupa en muchos ejemplos, y es uno de los principales principios de la programación funcional.

## Programación funcional en JavaScript

En base a lo explicado anteriormente, es necesario llevar a nuestro curso el por qué es importante hablar de **programación funcional**.

En primer lugar y como ya saben, JavaScript es un lenguaje con una definición de variables sencillas. Al ser sencillo de manipular, es posible que hayas notado que puedes definir variables como funciones, o funciones que reciben como parámetro otra función para retornar un valor o una función. Todo eso presente en JavaScript es propio o inherente de la programación funcional.

La programación funcional al poner como estrella a la función, permite además manipularla a antojo, generando recursividad, asignando variables a funciones, o funciones que retornan otras para generar callbacks sobre la misma. El concepto de programación funcional se verá en profundidad en el subcapítulo de funciones, en donde definiremos las **arrow functions**, concepto introducido en ES6+.

## Programación orientada a Eventos

Ya sabemos que JavaScript es orientado a objetos, posee cualidades del paradigma funcional pero se destaca, sobretodo en FrontEnd en una característica igual de importante. La programación orientada a eventos.

Es importante señalar que un evento y un objeto son cosas distintas. Un objeto como tal, es una representación abstracta de algo que puede ser descrito como algo concreto. Todo en sí es un objeto, un auto es un objeto, el auto tiene componentes que cada uno por sí solos son objetos. La atomicidad de estos objetos solo se limita a la imaginación del creador.

Los objetos poseen métodos y estructuras de datos abstractas. Volviendo al ejemplo, un auto posee una carrocería, un motor, componentes electrónicos, puertas, ruedas, (...). Y sus funcionalidades son avanzar o detenerse, acelerar o frenar.

Pero ahora nos cabe una duda ¿Quién o cuándo se gatillan dichos eventos o funcionalidades? ¿Quién es el encargado de hacer funcionar dicho objeto (auto)?

Esto se responde con la **Orientación a Eventos**, donde el usuario es finalmente el que determina cómo y cuándo se gatillas los eventos o funcionalidades de un objeto o inclusive, funciones que no pertenezcan a ningún objeto.

Los lenguajes orientados a Eventos, como lo es Visual Basic y JavaScript, están especialmente hechos para interactuar con el usuario mediante interfaces gráficas. Se puede decir que son lenguajes de Front End, es decir que su principal función está en la presentación de datos e interacción con el usuario. Por eso el que estén dirigidos a Eventos es algo necesario, eventos como el dar click en un botón, cambiar el valor de un checkbox, deben ser interpretados por el lenguaje. En cambio, lenguajes como PHP, están más orientados al Back End, ejecutándose en un servidor pueden funcionar de una manera más estructurada siguiendo una lógica secuencial. Su función está más en procesar los datos generados por el usuario.

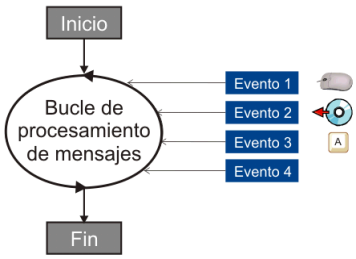


Imagen 1. Ejemplo de funcionamiento de programación funcional.

En la imagen 1, podemos apreciar que tenemos una máquina central que es capaz de encolar todos los eventos que puedan ser gatillados por usuarios a través de interfaces como mouse, teclado, discos, cualquier tipo de periférico como tal, los eventos quedan encolados y se generan salidas.

Ahora nos queda preguntarnos ¿cómo hace JavaScript para poder gatillar tantos eventos a la vez sin necesidad de que espere terminar uno para comenzar con el siguiente. Es acá donde introduciremos el concepto de asincronía (concurrencia) y procesos paralelos.

## Concurrencia y procesos paralelos

Concurrencia y paralelismo son conceptos relacionados pero con un importante matiz de diferencia entre ellos. Es por esto que, muy a menudo se confunden y se utilizan erróneamente. Vayamos al grano:

- Concurrencia: cuando dos o más tareas progresan simultáneamente.
- Paralelismo: cuando dos o más tareas se ejecutan, literalmente, a la vez, en el mismo instante de tiempo.

Nótese la diferencia: que varias tareas progresan simultáneamente no tiene porque significar que sucedan al mismo tiempo. Mientras que la concurrencia aborda un problema aún más general, el paralelismo es un sub-caso de la concurrencia donde las cosas suceden exactamente al mismo tiempo.

Mucha gente aún sigue creyendo que la concurrencia implica necesariamente más de un thread (Secuencia lineal). Esto no es cierto. El entrelazado (o multiplexado), por ejemplo, es un mecanismo común para implementar concurrencia en escenarios donde los recursos son limitados. Piensa en cualquier sistema operativo moderno haciendo multitarea con un único core. Simplemente corta las tareas, las hace más pequeñas y las entrelaza, de modo que cada una de ellas se ejecutará durante un breve instante. Sin embargo, a largo plazo, la impresión es que todas progresan a la vez.

Fíjate en el siguiente gráfico:

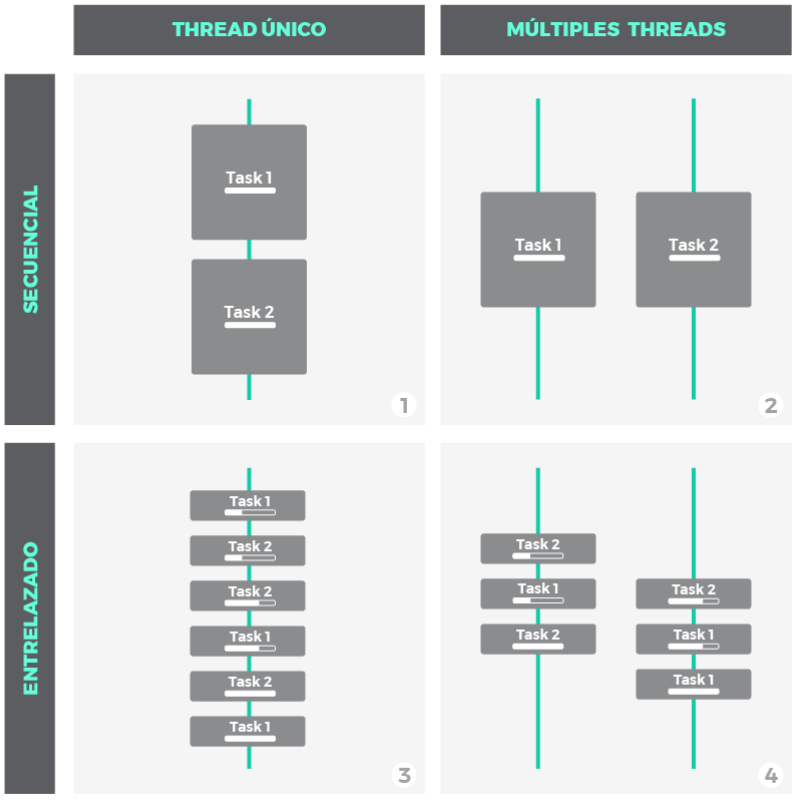


Imagen 2. Concurrencia y paralelismo representado en cajas de tareas.

- Escenario 1: no es ni concurrente ni paralelo. Es simplemente una ejecución secuencial, primero una tarea, después la siguiente.
- Escenario 2, 3 y 4: son escenarios donde se ilustra la concurrencia bajo distintas técnicas:
    - Escenario 3: muestra como la concurrencia puede conseguirse con un único thread. Pequeñas porciones de cada tarea se entrelazan para que ambas mantengan un progreso constante. Esto es posible siempre y cuando las tareas puedan descompuestas en subtareas más simples.
    - Escenario 2 y 4: ilustran paralelismo, utilizando multiples threads donde las tareas o subtareas corren en paralelo exactamente al mismo tiempo. A nivel de thread, el escenario 2 es secuencial, mientras que 4 aplica entrelazado.

Entendamos que *threads* o en español *hilos*, son secuencias lineales que se pueden configurar de forma automática o manual, ocupan los núcleos de tu procesador o núcleos virtuales para generar trabajos de forma paralela, sin depender unas de otras. Cuando existe una dependencia, el hilo queda en espera a que otro hilo tenga lista su tarea para seguir continuando, esto nos lleva al concepto de semáforo. En profundidad estos temas son complejos, por ende, nos limitaremos a que depende de la máquina que utilicemos, será la cantidad de procesos paralelos que podemos realizar.

JavaScript fue diseñado para ser ejecutado en navegadores, trabajar con peticiones sobre la red y procesar las interacciones de usuario, al tiempo que se mantiene una interfaz fluida. Ser bloqueante y síncrono no ayudaría a conseguir estos objetivos, es por ello que JavaScript ha evolucionado intencionadamente pensando en operaciones de tipo I/O. Por esta razón:

- JavaScript utiliza un modelo asíncrono y no bloqueante, con un loop de eventos implementado con un único thread para sus interfaces de entrada/salida.

Gracias a esta solución, JavaScript es áltamente concurrente a pesar de emplear un único thread. Ya conocemos el significado de asíncrono y no bloqueante, pero ¿qué es el loop de eventos? Este mecanismo será explicado ahora. Pero antes, a modo de repaso, veamos el aspecto de una operación I/O asíncrona en Javascript:

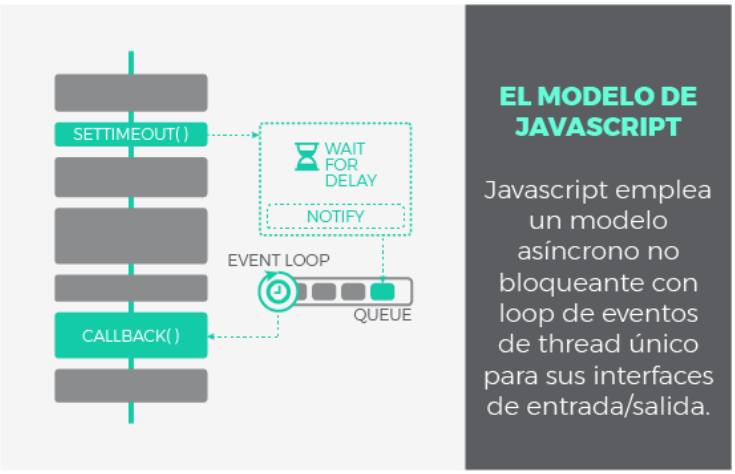


Imagen 3. Como trabaja Javascript sus procesos.

Explicaremos cada uno de dichos procesos con la siguiente imagen:


Imagen 4. Explicación de los procesos señalados en imagen 3.

Aunque JavaScript ha sido concebido con las operaciones de entrada/salida en mente, no significa que no pueda ejecutar tareas de procesado intensivo. Por supuesto que puede hacerlo, pero si no se manejan adecuadamente, podría dar lugar a los problemas mencionados en el apartado anterior.

Se ha invertido un considerable esfuerzo ultimamente para minimizar estos problemas. Como resultado, entidades como los WebWorkers y los SharedArrayBuffer han visto la luz recientemente para introducir el paralelismo en JavaScript. Si necesitas ejecutar tareas pesadas que hagan un uso intensivo de CPU deberías considerar el uso de WebWorkers que corran en segundo plano consumiendo threads distintos al principal.

### Callbacks

Los callbacks son la pieza clave para que JavaScript pueda funcionar de forma asíncrona. De hecho, el resto de patrones asíncronos en JavaScript está basado en callbacks de un modo u otro, simplemente añaden azúcar sintáctico para trabajar con ellos más cómodamente.

Un callback no es más que una función que se pasa como argumento de otra función, y que será invocada para completar algún tipo de acción. En nuestro contexto asíncrono, un callback representa el '¿Qué quieres hacer una vez que tu operación asíncrona termine?'. Por tanto, es el trozo de código que será ejecutado una vez que una operación asíncrona notifique que ha terminado. Esta ejecución se hará en algún momento futuro, gracias al mecanismo que implementa el bucle de eventos.

Fíjate en el siguiente ejemplo sencillo donde utilizamos un callback:

```
setTimeout(function(){
  console.log("Hola Mundo con retraso!");
}, 1000)
```

Si lo prefieres, el callback puede ser asignado a una variable con nombre en lugar de ser anónimo:

```
const myCallback = () => console.log("Hola Mundo con retraso!");
setTimeout(myCallback, 1000);
```

Acá lo podemos notar con ES6+.

`setTimeout` es una función asíncrona que programa la ejecución de un callback una vez ha transcurrido, como mínimo, una determinada cantidad de tiempo (1 segundo en el ejemplo anterior). Con este fin, dispara un timer en un contexto externo y registra el callback para ser ejecutado una vez que el timer termine. En resumen, retrasa una ejecución, como mínimo, la cantidad especificada de tiempo.

Dicho concepto te lo planteo con el objetivo de que lo conozcas. Cuando lleguemos a la definición de funciones, revisaremos otros ejemplos más complejos.

## Conceptos básicos: Introducción a ES6

Para entender por qué ES6+, primero es necesario señalar cuáles fueron los objetivos de los desarrolladores en generar dicho cambio de paradigma en JavaScript:

- Aplicaciones complejas.
- Bibliotecas (incluyendo el DOM) compartidas por esas aplicaciones.
- Generadores de código dirigidos a la nueva edición.
- Incorporación de módulos que antes se trabajaban de forma aparte, ahora son parte del lenguaje como tal (Programación funcional, promesas, async - await, entre otros).
- Clases: se basan en cómo se utilizan actualmente las funciones de constructor.
- Módulos: recoge ideas de diseño del formato del módulo CommonJS.

- Arrow function: tienen sintaxis que se toma prestada de CoffeeScript (considerado uno de los mejores legados de coffeescript).
- Parámetros de funciones con nombre: no hay soporte incorporado para los parámetros con nombre. En lugar, la práctica existente de nombrar parámetros a través de literales de objetos se admite a través de la desreferenciación.

Aunque pueda parecer que las principales características de ES6+ no son tantas, en su práctica, cada una de estas funcionalidades añadidas aborda un mundo a la hora de escribir código complejo.

## Conceptos básicos: Variables de ES6

En versiones conocidas, era sabido que las variables se declaraban con el prefijo **var**, como vemos en el siguiente ejemplo:

```
function explainVar(){
 var a = 10;
 console.log(a);  // output 10
 if(true){
  var a = 20;
  console.log(a); // output 20
 }
 console.log(a);  // output 20
}
```

En el ejemplo, **var** es una forma de declarar una variable de modo que se hace visible en todos los contextos, a su vez, si la variable es modificada en un contexto distinto, puede alterar los procesos. En este sentido, si tenemos una función que define `var a = 10`. Fuera de esa función **a** seguirá valiendo 10, si secuencialmente cambiamos dicho valor e invocamos nuevamente la función, pisará siempre el valor anterior.

### Var

Ya se explicó un poco como funciona `var`, ahora entenderemos un poco más la diferencia con la forma moderna de verlo con ES6+.

Es importante señalar que ahora `var`, desde la formalidad, se evita, puesto que se introducen nuevas formas de declarar variables, lo bueno de esto es que `var` se utiliza para casos puntuales en donde una variable puede mutar pero sin alterar el orden de ejecución de un programa en JavaScript. Siempre se debe tener presente que `var` es una forma lícita de almacenar datos de variables, pero si se usa la misma variable en diversas funciones que pueden ser invocadas de forma simultanea por eventos, es posible que altere los resultados.

Analicemos un ejemplo de `var`:

```
{
  var salutation = 'hi';
  console.log(salutation); // hi
  console.log(this.salutation); // hi
  function runModule() {
    var salutation = 'hola';
    if (1 == 1) { // esto siempre será TRUE
      console.log(salutation); // hola
```

```
    }
  }
  runModule();
  console.log(salutation); // hi
  if (1 == 1) {
    var salutation = 'cheers';
    console.log(salutation) // cheers
    if (2 == 2) {
      var salutation = 'cheers again'
      console.log(salutation) // cheers again
    }
    console.log(salutation) // cheers again
  }
  console.log(salutation); // cheers again
  salutation = 'bye'
  console.log(salutation); // bye
  console.log(this.salutation); // bye
}
```

Como se puede observar, `var` cambia según el contexto, sin embargo, es legible en todos los entornos, cuando se vuelve a definir `var` pisa el valor que tenía, independiente de donde fue declarado. Esta lógica cambia con el tipo de declaración `let`.

## Let

Por otra parte tenemos a `let`, esta es una nueva forma de definir variables en ES6+ y viene a complementar la forma de trabajar anteriormente con `var`. El alcance de estas variables, es que solo pueden ser accedidas dentro del bloque donde se definen. También, permiten que su valor pueda ser reasignado.

Veamos el siguiente ejemplo:

```
if(true){
    let helloWorld = "Hola mundo";
    helloWorld = "Hola mundo :)";
    console.log(helloWorld);
    //Hola mundo :)
}
console.log(helloWorld);
//ReferenceError: helloWorld is not defined
```

Como se puede observar, al definir una variable dentro de un bloque de código que no necesariamente se puede ingresar, `let` no está definido, puesto que fuera de dicho bloque de código, la variable no existe. Esto pasa porque `let` introduce el *"block scope"*. La variable asignada como `let` solo será accesible dentro del `for loop`.

Algunos otros ejemplos de `let`:

```
function explainLet(){
 let a = 10;
 console.log(a);  // output 10
 if(true){
  let a = 20;
  console.log(a); // output 20
 }
 console.log(a);  // output 10
}
```

Acá se puede observar claramente que según donde se genere la variable como tal, es el scope que abordará. Es decir, si defino y genero una nueva variable dentro de un bloque de código, esta va a ignorar lo que determina el exterior, sin embargo al cerrar dicho bloque de código, la variable volverá a tener el valor dado por el padre.

Acá otro ejemplo:

```
function explainLet(){
 let a = 10;
 let a = 20; //throws syntax error
 console.log(a);
}
```

Como es lógico, declarar 2 veces una misma variable de tipo let arrojará un error, puesto que esto no está permitido en ES6.

## Const

`Const` es un poco más sencillo de entender, puesto que opera excatamente igual que `let`, con la diferencia en que dentro de un scope, no se puede reasignar un valor.

Veamos un ejemplo:

```
function explainConst(){
 const x = 10;
 console.log(x); // output 10
 x = 20; //throws type error
 console.log(x);
}
```

Y es que claro, dentro de dicha función no es posible reasignarle un valor como tal.

```
const a = 50;
a = 60; // shows error. You cannot change the value of const.
const b = "Constant variable";
b = "Assigning new value"; // shows error.
//Consider another example.
const LANGUAGES = ['Js', 'Ruby', 'Python', 'Go'];
LANGUAGES = "Javascript"; // shows error.
LANGUAGES.push('Java'); // Works fine.
console.log(LANGUAGES); // ['Js', 'Ruby', 'Python', 'Go', 'Java']
```

A una constante le puedo añadir valores cuando se tratan de arreglos y objetos, lo que no puedo es pisarlos por un valor completamente distinto. Quizás pueda ser un poco dificil de entender, sin embargo todo se reduce a que el valor inicial sigue siendo el mismo, al hacer un push en un arreglo, claramente la varible cambió, pero no se perdió ningún dato anterior.

En mi opinión es ideal usar `const` si no tienes que reasignar el valor de tu variable en ningún momento. ¿Por qué? Bueno. Queremos minimizar el estado de mutación o "mutable state". Es importante saber que en nuestro código estarán corriendo muchos procesos a la vez. Cada que uses `let` es bueno analizar si realmente cambiará el valor de esta variable en algún momento. De no ser así, utiliza `const`.

En conclusión, siempre evita usar `var`, puesto que puede prestarse para confusiones y overriding de variables. En su lugar intenta usar `const` cuando sepas que asignas un valor que no cambiará con el tiempo y `let` cuando sepas que es necesario que la variable cambie con el tiempo pero que no se vea afectada en diversos scopes.

## Conceptos básicos: Funciones de ES6

En ES6, el concepto de funciones cambia, en este sentido es importante destacar que, aunque los cambios no se vean grandes, la capacidad que le entrega al lenguaje de realizar operaciones de forma mucho más eficiente crece significativamente. El lenguaje al tomar un camino bajo la lógica objeto-funcional, permite generar funciones que operan eficientemente de forma recursiva, sin la necesidad de integrar librerías externas que faciliten dichas operaciones.

Era común en versiones anteriores, usar Coffescript para suplir algunas falencias del lenguaje a nivel de servidor, o Lodash para añadir la capa de funcionalidad en donde JavaScript por sí solo quedaba corto, considerando además que las nuevas funcionalidades y las antiguas siempre apuntan a la asincronía natural que ofrece el lenguaje por sí mismo.

### Arrow function

Las funciones en ES6 cambian un poco en su sintaxis. Hay que dejar en claro que de igual forma se puede usar la sintaxis antigua pero existen cosas que cambien entre las 2 versiones:

```
// Old Syntax
function oldOne() {
 console.log("Hello World!");
}
// New Syntax
var newOne = () => {
 console.log("Hello World!");
}
```

La nueva sintaxis puede ser un poco confusa. Pero voy a tratar de explicarla.

Hay dos partes de la sintaxis.

```
var newOne = ()
=> {}
```

La primera parte es simplemente declarar una variable y asignarle la función (). Simplemente dice que la variable es en realidad una función.

Entonces, la segunda parte es declarar la parte del cuerpo de la función. La parte de la flecha define la parte del cuerpo.

Otro ejemplo con parámetros:

```
let NewOneWithParameters = (a, b) => {
 console.log(a+b); // 30
}
NewOneWithParameters(10, 20);
```

Como se puede observar en este ejemplo, se define una función asignada a una variable (ya lo mencionamos en la sección de programación funcional) se define que dicha variable es una función que recibe como parámetros a y b, y luego los suma.

La función puede ser invocada dentro de su scope.

Las arrow function se diferencian con el resto puesto que consideran cualquier variable, let, const o var fuera de su entorno como si estuvieran definidas dentro, la notación antigua de función forzaba a pasar dichos valores por parámetros. Por ende, las arrow function son mucho más completas puesto que pueden interpretar como propias las acciones que ocurren fuera de su propio scope. Es por esto que se debe tener cuidado con los cambios de variables al trabajar con var sobretodo.

## Parámetros default

Los parámetros default son parámetros que se dan de forma predeterminada al declarar una función. Pero, su valor se puede cambiar al llamar a la función.

Veamos un ejemplo:

```
let Func = (a, b = 10) => {
 return a + b;
}
Func(20); // 20 + 10 = 30
```

En el ejemplo anterior, estamos pasando solo un parámetro. La función hace uso del parámetro predeterminado y ejecuta la función.

Veamos otro ejemplo de invocación de la misma función:

```
Func(20, 50); // 20 + 50 = 70
```

Como ahora le pasamos el segundo parámetro, ya no toma como valor default el definido en la función puesto que se pasa el parámetro con un valor distinto al default.

Este tipo de forma de escribir una función permite validar al recibir los parámetros si vienen y con qué valor vienen.

Cuando se llama a la función con parámetros, se asignan en el orden. Es decir, el primer valor se asigna al primer parámetro y el segundo se asigna al segundo parámetro, etc.

## Callbacks

En secciones anteriores ya hablamos de callbacks, ahora nos centraremos en analizar ejemplos:

```
function functionTwo(var1, var2, callback1, callback2) {
    callback1(var1);
    callback2(var2);
}

functionTwo(1, 2, function (x) { alert(x); }, function (x) { alert(x); })
```

Como se puede observar en el ejemplo, tenemos una función que recibe 4 parámetros, 2 de ellos son callbacks, al momento de invocar la función, esta recibe como parámetros valores que serán invocados en la misma y se gatillarán de forma asíncrona.

Te recomiendo para entender el concepto, que pruebes ejecutando la función para ver que hace.

## Strings

Una de las principales características que tienen los lenguajes modernos es la capacidad de computar y trabajar los `strings`. En este sentido, la nueva versión de ES6 profundiza algunos elementos que no existían en versiones anteriores, generando los siguientes resultados:

### Includes

Si antes queríamos verificar si un `string` tenia una arroba teníamos que crear una expresión regular o realizar un bucle para matchear ese valor. `Includes` nos soluciona ese problema de una forma muy declarativa y simple.

Por lo tanto, si quisiéramos saber si una cadena tiene una cadena de texto podríamos realizar lo siguiente:

```
let email = 'some@text.com';
if (email.includes('@'))
 console.log('Email válido!');
else
 console.log('Email no válido!');
// ---> Email válido!
```

Como podrás observar, la función `includes` nos devuelve true o false, por lo tanto, al insertarlo dentro de un condicional nos ahorramos tener que guardar su valor en una variable. La función `includes` recibe como argumento el carácter a buscar. En este caso, no es necesario utilizar llaves en el condicional, porque si recordamos, JavaScript hereda de C y una de las características es que si vamos a ejecutar una sola instrucción no es necesario la creación de un bloque de código.

**startsWith**

La función `startsWith`, como su nombre lo indica, nos sirve para verificar si una cadena en cuestión comienza con X caracter.

Veamos un ejemplo:

```
let email = 'some@text.com';
if (email.startsWith('some'))
 console.log('El Email comienza con "some"!');
else
 console.log('El Email no comienza con "some"!');
// ---> El Email comienza con "some"!
```

**Repeat**

La función `repeat` nos permite repetir una cadena de texto una cierta cantidad de veces.

```
let str = 'JavaScript! ';
console.log(str.repeat(3));
// ---> JavaScript! JavaScript! JavaScript!
```

Tenemos una variable con un String y mediante la función repeat le pasamos la cantidad de veces que vamos a repetir el texto.

Eso es todo en esta sección, aún quedan muchos conceptos que explorar de JavaScript ES6. La cantidad de funciones añadidas es mucho más amplia pero se verá en profundidad más adelante. **ES6+ tiene una característica y es que las funciones son pensadas para ser aplicadas a objetos**. Por ende, la gran mayoría de las funciones que se analizarán en la siguiente unidad serán pensadas para poder ser utilizadas en arreglos, objetos, strings.