Clase 3 - JavaScript

En JavaScript nosotros podemos declarar una funcion con la palabra reservada function. Las funciones en JavaScript no especifican su retorno al ser debilmente tipado. Todas las funciones de JS devuelven algo, si no utilizamos return se devuelve undefined.

Cuando utilizamos function la funcion es declarada.

- Ejemplo: function saludar(){return "Hola";}

Podemos hacer una funcion expresada si declaramos una funcion y se la asignamos a una variable. En general utilizamos constantes. Se llama expresada porque es resultado de una expresion. La variable contiene la referencia a la funcion, la direccion de memoria de donde esta el codigo asignado.

- Ejemplo: const unaFuncion = function despedir (){return "Nos vemos";}.

Podemos declarar una variable declarada y despues asignarla a una constante, haciendo posible la ejecucion de ambas. En cambio si declaro una expresada no puedo ejecutar la funcion asociada/la funcion referenciada. Si o si utilizamos el nombre de la constante.

- const unaFuncion = function despedir(){ console.log("Chau nos vemos");}
  - unaFuncion(); -> Se ejecuta
  - despedir(); -> No se ejecuta, tenemos que utiliza la constante que la referencia.
- function despedir(){console.log("Chau nos vemos");}
  - const unaFuncion = despedir;
  - despedir(); -> Se ejecuta
  - unaFuncion(); -> Se ejecuta

Las funciones anonimas son funciones asociadas a variables que no poseen nombres en su declaracion. La constante guarda la direccion de memoria.

- Ejemplo: const unaFuncionTres = function (){ console.log("Chau nos vemos");}

Las arrow function/funcion lambda son funciones anonimas que se asocian con una constante. No utiliza la palabra function y en su declaracion utiliza (PARAMETROS) para recibir parametros => para iniciar la declaracion y {RETURN ALGO/EJECUTAR ALGO} para el cuerpo.
Si tenemos un solo parametro podemos obviar los (), si es de una sola linea podemos obviar los {} ya que su return es implicito.

- Ejemplo:
  - const unaFuncionCuatro = n => `Chau ${n}`;
  - const unaFuncionCinco = (x, y) => {return x * y;}
    - console.log(unaFuncionCuatro("Jose"));
    - console.log(unaFuncionCinco(10, 2));

Los back ticks `` los utilizamos para template strings.

Tanto las funciones expresadas como las funciones anonimas y las arrow function las guardamos en constantes. Se utilizan constantes ya que durante el tiempo de ejecucion se convierten en variables de solo lectura y no pueden cambiar, aparte al intentar cambiarlo en tiempo de diseño nos advierte.

var no se utiliza mas post 2015, se sigue utilizando por un tema de retrocompatibilidad. var tiene un ambito global sin importar el scope donde fue declarado. No tienen ambito de bloque tienen ambito de funcion. Tambien tiene un problema: puede ser re declarado la misma variable X cantidad de veces sin importar el tipo.

let si va a vivir dependiendo del scope donde fue declarado, tampoco permite que se sobredeclare la variable si el scope entra en conflicto.

const vive en ambito de bloque y no puede volver a cambiarse en un futuro, convirtiendose en una variable de solo lectura, si queremos cambiarlo nos rompe el programa. Las utilizamos tanto para los arrays, objetos y constantes, ya que nos devuelve la referencia y esa referencia nunca cambia. Todo lo que no sea tipo de dato nativo es object. Con let es muy facil des-referenciar objetos, lo que lo vuelve muy peligroso en desarrollos a gran escala, por ello utilizamos const como buena practica.

En JavaScript no hay sobrecargas como en C# o Java. Si tenemos 3 parametros y pasamos 2 argumentos va a recibir los parametros y al tercero lo va a tratar como un undefined. Esto puede provocar problemas si esperabamos las tres variables si o si. Para que esto no ocurra hay que validar todo en las funciones.
El == verifica que el contenido sea igual sin importar el tipo, el === va a verificar el contenido y el tipo de dato.

El spread operator/operador de propagacion nos permite acceder a todos los valores de un array. Sirve por ejemplo para copiar un array dentro de otro.
Enviar valores de un array a una funcion, enviar X cantidad de valores a una funcion independientemente de sus parametros es llamado rest parameters.

- Ejemplo:
  - const numeros = [3,4,5,6,7];
  - const numerosCopiados = [...numeros];
- Tambien podemos agregar mas numeros independientes al array
  - const numerosCopiados = [...numeros,10,20];

Se llaman delegados en C# porque delega la ejecucion y responsabilidad de una función a otra funcion. En JS llamamos a estos delegados callbacks.

Una API es un conjunto de funciones que me permite interactuar con algo. En el navegador hay distintas API's, el API de los timers que hace uso setTimeout, el API del DOM, etc.

En JavaScript hay una función que se llama setTimeout(). Esta función es asincrona, sirve como temporizador. Recibe 2 parametros, el primer parametro es una funcion anonima/referencia a funcion y el segundo parametro es el timepo en milisegundos a esperar.
El navegador por la arquitectura cliente-servidor tiene una API de los timers que nos permite estacionar/esperar a la funcion el tiempo dado en el timer y así liberar el stack de llamadas. Mientras se ejecuta el resto del stack la funcion espera el tiempo del timeout en la API, luego esa ejecucion de la funcion va al callback queue que ejecuta esa funcion en la pila. Para ejecutar que el callback ejecute en el stack la funcion el stack debe estar liberado.

De la misma manera funciona el setInterval, solo que hace el procedimiento de esperar X cantidad de milisegundos repitiendose.

El manejador de eventos en JavaScript se basa en callbacks para ejecutar un evento. Ese evento se carga en la API a la espera de ser ejecutado, asi la API de los eventos evita bloquear el stack, similar al API de los timers.
La funcion asociada a un evento (ya cargada en la API de eventos) va al callback queue cuando se lanza y espera a la liberacion del stack para ser ejecutado.