KSQUARE INGRESO

JS

Que es Javascript?

JavaScript is a programming language that is either interpreted or compilated just-in-time, with first-class functions. Despite being an scripting language (command sequenced) for web pages, it is actually a language used for different environments outside the web browser, such as Node.js for Server-side applications, Apache CouchDB and even Adobe Acrobat.

JS is a programming language based on prototypes, multiparadigm, single-threaded, dynamic typed and supports OOP.

POO Principles

Inheritance

Polymorphism

Abstraction

Encapsulation

What are the differences between "var" and "let"

“var” has no block scope

Variables, declared with var, are either function-scoped or global-scoped. They are visible through blocks.

As var ignores code blocks, we’ve got a global variable test.

```
if (true) {
  var test = true; // use "var" instead of "let"
}

alert(test); // true, the variable lives after if
```

The same thing for loops: var cannot be block- or loop-local:

```
for (var i = 0; i < 10; i++) {
  var one = 1;
  // ...
}
```

```
alert(i);   // 10, "i" is visible after loop, it's a global variable
alert(one); // 1, "one" is visible after loop, it's a global variable
```

IMPORTANT!: If a code block is inside a function, then var becomes a function-level variable:

```
function sayHi() {
  if (true) {
    var phrase = "Hello";
  }

  alert(phrase); // works
}

sayHi();
alert(phrase); // ReferenceError: phrase is not defined
```

“var” tolerates redeclarations

With var, we can redeclare a variable any number of times. If we use var with an already-declared variable, it’s just ignored:

```
var user = "Pete";

var user = "John"; // this "var" does nothing (already declared)
// ...it doesn't trigger an error

alert(user); // John
```

If we declare the same variable with let twice in the same scope, that’s an error:

```
let user;
let user; // SyntaxError: 'user' has already been declared
```

“var” variables can be declared below their use

var declarations are processed when the function starts (or script starts for globals).

In other words, var variables are defined from the beginning of the function, no matter where the definition is (assuming that the definition is not in the nested function).

So this code:

```
function sayHi() {
  phrase = "Hello";
```

```
  alert(phrase);

  var phrase;
}
sayHi();
```

...Is technically the same as this (moved var phrase above):

```
function sayHi() {
  var phrase;

  phrase = "Hello";

  alert(phrase);
}
sayHi();
```

...Or even as this (remember, code blocks are ignored):

```
function sayHi() {
  phrase = "Hello"; // (*)

  if (false) {
    var phrase;
  }

  alert(phrase);
}
sayHi();
```

People also call such behavior "hoisting" (raising), because all var are "hoisted" (raised) to the top of the function.

So in the example above, if (false) branch never executes, but that doesn't matter. The var inside it is processed in the beginning of the function, so at the moment of (*) the variable exists.

Summary

There are two main differences of var compared to let/const:

1. var variables have no block scope, their visibility is scoped to current function, or global, if declared outside function.
2. var declarations are processed at function start (script start for globals).

What is Hoisting?

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

Declarations are hoisted, but assignments are not.

That's best demonstrated with an example:

```
function sayHi() {
  alert(phrase);

  var phrase = "Hello";
}

sayHi();
```

The line var phrase = "Hello" has two actions in it:

Variable declaration var Variable assignment =.

The declaration is processed at the start of function execution ("hoisted"), but the assignment always works at the place where it appears. So the code works essentially like this:

```
function sayHi() {
  var phrase; // declaration works at the start...

  alert(phrase); // undefined

  phrase = "Hello"; // ...assignment - when the execution reaches it.
}

sayHi();
```

Because all var declarations are processed at the function start, we can reference them at any place. But variables are undefined until the assignments.

In both examples above, alert runs without an error, because the variable phrase exists. But its value is not yet assigned, so it shows undefined.

Function Declaration vs Function Expression

In JS we can create functions with two different methods, Function Declaration and Function Expression.

Function declaration

This method uses the keyword function followed by the name of the function, this method allows to declare the function and use it even when it was declared after it's use. Let's see an example.

```
// Function Declaration

obtenerCliente("Juan Pablo");

function obtenerCliente(nombre) {
    console.log(`El nombre del cliente es: ${nombre}`)
}

// output -> El nombre del cliente es Juan Pablo
```

In this example we can use the function before it's declaration, that's because JS first register the functions declared and then it registers the variables.

Function Expression

This method assigns a function to a variable, and it cannot be used before it is declared.

```
// Function Expression

const obtenerCliente2 = function (nombre) {
    console.log(`El nombre del cliente es: ${nombre}`)
}

obtenerCliente2("Dante");
```

If we try to call the function before it's definition, JS will throw an error saying there's no such function defined in our program.

What is recursivity?

With respect to a programming function, recursion happens when a function calls itself within its own definition. It calls itself over and over again until a base condition is met that breaks the loop.

There are 2 main parts of a recursive function; the base case and the recursive call. The base case is important because without it, the function would theoretically repeat forever (in application there would be what is referred to as a “stack overflow” to stop the repetition which we will touch on a little later). Below is an example of a simple recursive function.

```
function getFactorial(number) {
    if(number === 1){
        return 1;
    }
    return number * getFactorial(number - 1);
}

console.log(getFactorial(5)); // Output: 120
```

What is the difference between Double equal (==) and triple equal sign (===)

When using triple equals === in JavaScript, we are testing for strict equality. This means both the type and the value we are comparing have to be the same.

When using double equals == in JavaScript we are testing for loose equality. Double equals also performs type coercion.

Type coercion means that two values are compared only after attempting to convert them into a common type.

Falsy Values

Okay, so why does false == 0 in JavaScript? It's complex, but it's because in JavaScript 0 is a falsy value.

Type coercion will actually convert our zero into a false boolean, then false is equal to false.

There are only six falsy values in JavaScript you should be aware of:

- false — boolean false
- 0 — number zero
- "" — empty string
- null
- undefined
- NaN — Not A Number

Triple Equals is superior to double equals. Whenever possible, you should use triple equals to test equality. By testing the type and value you can be sure that you are always executing a true equality test.

Tipos de datos - Primitivos

Los tipos de datos primitivos en JavaScript **son aquellos que no poseen métodos ni propiedades**. Además, los valores asignados con estos tipos de datos son inmutables, lo que quiere decir que después de asignar una variable a un valor primitivo, si deseas cambiar su valor necesitaras reasignarle un valor nuevo, ya que su valor inicial no puede ser modificado, simplemente se substituye con el nuevo valor.

Tenemos siete (7) tipos primitivos en JavaScript por el momento:

- string
- number
- boolean
- null
- undefined
- Symbol
- bigint.

Es JavaScript un lenguaje orientado a objetos (OPP)? No, JavaScript es un lenguaje basado en prototipos

JavaScript sólo tiene una estructura: **objetos**. Cada objeto tiene una propiedad privada (referida como su [[Prototype]]) que mantiene un enlace a otro objeto llamado su prototipo. Ese objeto prototipo tiene su propio prototipo, y así sucesivamente hasta que se alcanza un objeto cuyo prototipo es **null**. Por definición, null no tiene prototipo, y actúa como el enlace final de esta cadena de prototipos.

Entonces en JavaScript tenemos el objeto global **Object** y la propiedad Object.prototype representa al objeto prototipo de Object, donde todos los objetos vienen de Object donde heredan las propiedades de Object.prototype.

> TODOS los objetos en JavaScript tienen su propio prototype.

> Los métodos y propiedades de Object.prototype son la base de cualquier objeto en JavaScript.

What are callback functions

Una función callback es aquella que es pasada como argumento a otra función para que sea "llamada de nuevo" (call back) en un momento posterior. Una función que acepta otras funciones como argumentos es llamada función de orden-superior (High-Order), y contiene la lógica para determinar cuándo se ejecuta la función callback. Es la combinación de estas dos la que nos permite ampliar nuestra funcionalidad.

¿Por qué usar funciones Callback?

La mayoría del tiempo estamos creando programas y aplicaciones que operan en una forma sincrónica. En otras palabras, algunos de nuestra operaciones comienzan solo después de que se hayan completado las anteriores. Usualmente, cuando solicitamos datos desde otras fuentes como una API externa, no siempre sabemos cuando nuestros datos serán devueltos. En estos casos queremos esperar la respuesta, pero no queremos que toda nuestra aplicación se detenga mientras se recuperan los datos. Estas son situaciones dónde las funciones callback resultan útiles.

What is a Closure?

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

To use a closure, define a function inside another function and expose it. To expose a function, return it or pass it to another function.

The inner function will have access to the variables in the outer function scope, even after the outer function has returned.

Among other things, closures are commonly used to give objects data privacy. Data privacy is an essential property that helps us program to an interface, not an implementation. This is an important concept that helps us build more robust software because implementation details are more likely to change in breaking ways than interface contracts.

In JavaScript, closures are the primary mechanism used to enable data privacy. When you use closures for data privacy, the enclosed variables are only in scope within the containing (outer) function. You can’t get at the data from an outside scope except through the object’s privileged methods. In JavaScript, any exposed method defined within the closure scope is privileged.

Take into consideration the properties prefixes, in Vanilla JS we can use an underscore(_) or a hashtag (#) to indicate that a property is private. In TS we can use the keywords private and protected.

Formas de crear objetos

- Object Literals
- Prototype
- Function constructors

Objects literals

Crear un objeto literalmente con sus propiedades y este asignarlo a una variable.

Ejemplo:

```
const person = {
    name: '',
    age: 0,
    presented() {
        return `Hello my name is ${this.getName()} and I am ${this.getAge()} years
old`;
    }
};
person.name = 'Yhoshua Ochoa';
person.age = 24;
console.log(person.presented()); //output Hello my name is Yhoshua Ochoa and I am
24 years old
```

Si observamos tendríamos que tener para cada persona el objeto completo, y si quisiéramos cambiar una propiedad o método, uffffff, creo que ha quedado claro que no es lo mejor... por suerte en JavaScript existen formas de hacerlo sin que tengamos que hacer nada raro.

Prototype

Otra forma de crear objetos (que no soluciona nuestro problema anterior), es mediante el prototipo del mismo, veamos el ejemplo anterior aplicado a Prototype:

```
const person = new Object();
person.name = 'Yhoshua Ochoa';
person.age = 24;
person.presented = function () {
    return `Hello my name is ${this.name} and I am ${this.age} years old`;
}
console.log(person.presented()); //output Hello my name is Yhoshua Ochoa and I am
24 years old
```

Si observamos el ejemplo, vamos a tener el mismo problema que con Object Literals de tener que asignar uno por uno a cada objeto sus propiedades, para este caso lo hacemos directamente a las propiedades al

prototipo, otra desventaja que tiene es que cuando el objeto tiene muchos métodos y propiedades es un poco ilegible al momento de actualizarlo o saber que hace.

Function constructors

Por fortuna existe otra forma (esta sí soluciona nuestro problema anterior) es mediante una función que crea un objeto a partir de un prototipo vacío.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.getName = function () {
        return this.name;
    }
    this.getAge = function () {
        return this.age;
    }
    this.presented = function () {
        return `Hello my name is ${this.getName()} and I am ${this.getAge()} years
old`;
    }
}
const person1 = new Person('Yhoshua Ochoa', 24);
console.log(person1.presented()); //output Hello my name is Yhoshua Ochoa and I am
24 years old
```

Primero se ha creado una función a través de una Function statement, dentro de esta función estamos utilizando la palabra reservada this (mas adelante veremos a fondo), donde se agregan las propiedades name, age así como el método presented.

El truco está en la palabra reservada new, en palabras de Bruce Burton:

> Cuando anteponemos la palabra clave new, JS se encarga primero de crear un objeto vacío, es decir, cuando ejecutamos el operador (función) new, se crea un nuevo objeto el cual va a envolver (Lexical environment) la función que se invoca después de esta palabra y cuando termina de ejecutarse la función en este caso Person(), new, se encarga de devolver el objeto pero con las definiciones que le dimos a this. Si no anteponemos new a nuestra función veremos que nuestras variables valen undefined, por lo tanto si tratase de usar la propiedad name, de dicho objeto mandaría un error.

La ventaja que tenemos con esto, es que podemos generar n instancias, y con esto aprovechamos la reutilización de código, y ahora sí un cambio en la función afectaría a todos aquellos que estén haciendo la instancia de objeto.

Object.create()

Otra forma de crear objetos es utilizando el metodo create() del prototipo Object, el cual se encargara de crear un objeto usando el utilizando el prototipo de un objeto existente:

Ejemplo:

```
const coder = {
    isStudying : false,
    printIntroduction : function(){
        console.log(`My name is ${this.name}. Am I studying?:
${this.isStudying}`);
    }
};
const me = Object.create(coder);
me.name = 'Mukul';
me.isStudying = true;
me.printIntroduction();
```

Clases

Tambien se pueden crear objetos a partir de clases, aunque JS en el fondo sigue utilizando la cadena de prototipos para manejar clases. De esta manera las clases en JS son azucar sintactico.

What is a promise?

What is destructuring?

What is Spread Operator?

What is a shadow copy?

What is Async / Await?

What is the Lexical Scope?

What "this" means in JS?

What is Big O notation?

How Classes work?

Bad practices with Objects and mutations (Object pollution) (Shadow copies)

Prototypes

Cohercion en JS

GIT

What is git rebase?

How to handle conflicts?

What is git cherry pick?