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

Introduction to Object Oriented Programming in JavaScript

Abstraction in JS

OOPS in JavaScript with easy to understand examples 🔥🔥

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.

What is the scope?

Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript. The two types of scope are local and global:

- **Global variables** are those declared outside of a block
- **Local variables** are those declared inside of a block

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
- Object Constructor
- 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.

Object Constructor

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?

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation was completed successfully.
- rejected: meaning that the operation failed.

The eventual state of a pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options occur, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

A promise is said to be settled if it is either fulfilled or rejected, but not pending.

MDN Promises

What is destructuring?

Destructuring is a JavaScript expression that allows us to extract data from arrays, objects, and maps and set them into new, distinct variables. Destructuring allows us to extract multiple properties, or items, from an array at a time.

1. Assigning to existing variable names

Here's how to destructure values from an object:

```
var employee = {    // Object we want to destructure
    firstname: 'Jon',
    lastname: 'Snow',
    dateofbirth: '1990'
};

// Destructuring the object into our variables
var { firstname, lastname, dateofbirth } = employee;
console.log( firstname, lastname, dateofbirth);
```

2. Assigning to new variable names

The following code destructures the object into variables with a different name than the object property:

```
var employee = {    // Object we want to destructure
    firstname: 'Jon',
    lastname: 'Snow',
    dateofbirth: '1990'
};

// Destructuring the object into variables with
// different names than the object variables
var { firstname: fn, lastname: ln, dateofbirth: dob } = employee;
console.log( fn, ln, dob);
```

3. Assigning to a variable with default values

We can also assign default values to variables whose keys may not exist in the object we want to destructure. This will prevent our variable from having an undefined value being assigned to it. The code below demonstrates this:

```
var employee = {    // Object we want to destructure
    firstname: 'Jon',
    lastname: 'Snow',
    dateofbirth: '1990'
};

// Destructuring the object into variables without
// assigning default values
var { firstname, lastname, country } = employee;
```

```
console.log("Without setting default values")
console.log( firstname, lastname, country);

// Destructuring the object into variables by
// assigning default values
var { firstname = 'default firstname',
      lastname = 'default lastname',
      country = 'default country' } = employee;
console.log("\n After setting default values")
console.log( firstname, lastname, country);
```

What is Spread Operator?

The spread operator is a new addition to the set of operators in JavaScript ES6. It takes in an iterable (e.g an array) and expands it into individual elements.

The spread operator is commonly used to make shallow copies of JS objects. Using this operator makes the code concise and enhances its readability.

Syntax

The spread operator is denoted by three dots, ...

1. Copying an array

The array2 has the elements of array1 copied into it. Any changes made to array1 will not be reflected in array2 and vice versa.

If the simple assignment operator had been used then array2 would have been assigned a reference to array1 and the changes made in one array would reflect in the other array which in most cases is undesirable.

```
let array1 = ['h', 'e', 'y'];
let array2 = [...array1];
console.log(array2);
```

2. Inserting the elements of one array into another

It can be seen that the spread operator can be used to append one array after any element of the second array. In other words, there is no limitation that baked_desserts can only be appended at the beginning or the end of the desserts2 array.

```
let baked_desserts = ['cake', 'cookie', 'donut'];
let desserts = ['icecream', 'flan', 'frozen yoghurt', ...baked_desserts];
console.log(desserts);
//Appending baked_desserts after flan
let desserts2 = ['icecream', 'flan', ...baked_desserts, 'frozen yoghurt'];
console.log(desserts2);
```

3. Array to arguments

```
function multiply(number1, number2, number3) {
  console.log(number1 * number2 * number3);
}
let numbers = [1,2,3];
multiply(...numbers);
```

Instead of having to pass each element like numbers[0], numbers[1] and so on, the spread operators allows array elements to be passed in as individual arguments.

```
//Passing elements of the array as arguments to the Math Object
let numbers = [1,2,300,-1,0,-100];
console.log(Math.min(...numbers));
```

The Math object of Javascript does not take in a single array as an argument but with the spread operator, the array is expanded into a number or arguments with just one line of code.

What is a shallow copy?

A shallow copy of an object is a copy whose properties share the same references (point to the same underlying values) as those of the source object from which the copy was made. As a result, when you change either the source or the copy, you may also cause the other object to change too — and so, you may end up unintentionally causing changes to the source or copy that you don't expect. That behavior contrasts with the behavior of a deep copy, in which the source and copy are completely independent.

What is Async / Await?

An async function is a function declared with the async keyword, and the await keyword is permitted within it. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Async functions may also be defined as expressions.

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: "resolved"
```

```
}

asyncCall();
```

Return Value

A Promise which will be resolved with the value returned by the async function, or rejected with an exception thrown from, or uncaught within, the async function.

Description

Async functions can contain zero or more await expressions. Await expressions make promise-returning functions behave as though they're synchronous by suspending execution until the returned promise is fulfilled or rejected. The resolved value of the promise is treated as the return value of the await expression. Use of async and await enables the use of ordinary try / catch blocks around asynchronous code.

> Note: The purpose of async/await is to simplify the syntax necessary to consume promise-based APIs. The behavior of async/await is similar to combining generators and promises.

Code after each await expression can be thought of as existing in a .then callback. In this way a promise chain is progressively constructed with each reentrant step through the function. The return value forms the final link in the chain.

What "this" means in JS?

In most cases, the value of this is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called. ES5 introduced the bind() method to set the value of a function's this regardless of how it's called, and ES2015 introduced arrow functions which don't provide their own this binding (it retains the this value of the enclosing lexical context).

const test = { prop: 42, func: function() { return this.prop; }, };

console.log(test.func()); // expected output: 42

What is the difference between Regular Functions and Arrow Functions?

The Difference Between Regular Functions and Arrow Functions

What is Big O notation?

Big-O is a standard mathematical notation that shows how efficient an algorithm is in the worst-case scenario relative to its input size. To measure the efficiency of an algorithm, we need to consider two things:

- **Time Complexity**: How much time does it take to run completely?
- **Space Complexity**: How much extra space does it require in the process?

Big-O notation captures the upper bound to show how much time or space an algorithm would require in the worst case scenario as the input size grows. It is usually written as:

f(n)=O(inputSize)

How Classes work?

MDN Classes

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

Javascript Worst Practice

Buggy JavaScript Code: The 10 Most Common Mistakes JavaScript Developers Make

What are some of the bad practices Javascript developers often do?

Prototypes

MDN Object Prototypes

What is JS Coercion?

There are two types of type coercion in JS, implicit and explicit.

Implicit coercion

Implicit coercion refers to the action that JS performs when you are operating on a variable, JS compiles the instruction and tries to convert the type of a variable into an available type for the operation to not throw an error.

For example:

```
const number1 = 10;
const number2 = "10";

const result = number1 + number2; // result: 1010
```

JS converts the type of number1 into a string, and then it concatenates the number2. This is implicit type coercion, the same happens if we compare the values using double equals operator (==).

Explicit coercion

Explicit coercion refers to the action of converting the type of a variable in an explicit way. In most cases this is performed when we use a function to convert the type of a variable into another type.

For example:

```
const number1 = "20";
const number2 = 20;

const convertedString1 = Number(number1); // value: 20, type: Number
const convertedNumber1 = number2.toString(); // value "20", type: String
```

In this case we are telling JS the type that we want to convert one variable into, this is called explicit coercion.

Type Coercion

What is Object Binding?

Object Binding refers to how JS will make a reference when you are using the "this" keyword. Object binding will assign an object to be used as the reference for the "this". This can be achieved in two different ways. Using implicit binding and using explicit binding.

Implicit binding

Implicit binding occurs when you create a new object using Object Literals and the "new" keyword with Function Constructor or classes.

For example:

```
const obj1 = {
  name: "Juan Pablo",
  age: 28,
  getInfo() {
    console.log(`Hello, my name is ${this.name} and i'm ${this.age} years old`)
  }
}

obj1.getInfo(); // "Hello my name is Juan Pablo and i'm 28 years old"
```

In this example JS can infer that the "this" will make a reference to obj1, it means we are not telling JS where this.name and this.age exist, it will automatically know that the reference is obj1. This is implicit binding.

Explicit Binding

Explicit binding will tell JS exactly what object will be used as reference for the "this" keyword. To accomplish Explicit Binding we can use JS built-in methods Bind(), Call() and Apply().

But they behave in different ways.

- Bind Bind will create a new function that will be assigned to a variable so it can execute a function with an specific object to be binded.

For example:

```
const obj1 = {
  name: "Juan Pablo"
}

const obj2 = {
  name: "Dante"
}
```

```
const getInfo = (age) => {
  console.log(`Hello, my name is ${this.name} and i'm ${age} years old`)
}

const bindedFunction1 = obj1.bind(getInfo, 28);
bindedFunction1() // "Hello my name is Juan Pablo and i'm 28 years old"

const bindedFunction2 = obj2.bind(getInfo, 35);
bindedFunction2() // "Hello my name is Dante and i'm" 35 years old"
```

In this example we are creating two objects with a name property. Then we create a function that uses this.name and receives a paramater age.

With bind we create two new functions that will bind the functions we declare to the object we want to be taken as reference for the "this", then we can execute them and they will know exactly what is the object to be used as reference for "this".

Note that the parameters for the bind() function will be the function to be binded and then the parameters, if there's more than 1 paramater to be used in the function they have to be passed as arguments one by one:

```
const bindedFunction = obj.bind(fn, arg1, arg2, arg3, ...)
```

- Call Call is also a method that can be used to bind a function to an object, but instead of creating a new function, the Call method will execute the function inmediatly. The parameters for the Call() function are exacly the same as Bind(), where each parameter of the function needs to be passed as individual arguments.

For example:

```
const obj1 = {
  name: "Juan Pablo"
}

const getInfo = (age) => {
  console.log(`Hello my name is ${this.name} and i'm ${age} years old`);
}

obj1.call(getInfo, 28); // "Hello my name is Juan Pablo and i'm 28 years old"
```

As we can see if we use the Call() method on an object, it will take the object as reference for the "this" and the parameters of the function will be passed down as individual arguments for the function execution, and then the function will be executed.

- Apply Apply() is rather similar to Call(), the only difference is that the arguments for the function will be passed as an array.

For example:

```
const obj1 = {
  name: "Juan Pablo"
}

const getInfo = (age, genre) => {
  console.log(`Hello my name is ${this.name} and i'm ${age} years old. I like
${genre} music`);
}

const info = [28, "Rock"]

obj1.call(getInfo, info); // "Hello my name is Juan Pablo and i'm 28 years old. I
like Rock music"
```

As we can see in the previous examples, JS allows us to specify the object to use as the "this" reference, we bind a function to an object in a explicit way. This is called explicit binding.

What are JS Modules?

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain the code-base.

JavaScript modules rely on the import and export statements.

Export

You can export a function or variable from any file.

Let us create a file named person.js, and fill it with the things we want to export.

There are two types of exports: Named and Default.

- Named Exports

You can create named exports two ways. In-line individually, or all at once at the bottom.

In-line individually:

person.js

```
export const name = "Jesse";
export const age = 40;
```

All at once at the bottom:

person.js

```
const name = "Jesse";
const age = 40;

export {name, age};
```

- Default exports

Let us create another file, named message.js, and use it for demonstrating default export.

You can only have one default export in a file.

message.js

```
const message = () => {
const name = "Jesse";
const age = 40;
return name + ' is ' + age + 'years old.';
};

export default message;
```

Import

You can import modules into a file in two ways, based on if they are named exports or default exports.

Named exports are constructed using curly braces. Default exports are not.

- Import from named exports

Import named exports from the file person.js:

```
import { name, age } from "./person.js";
```

- Import from default exports

Import a default export from the file message.js:

```
import message from "./message.js";
```

Error-First Callback in Node.js

Is a function which either returns an error object or any successful data returned by the function.

What is REST

Use Strict

Use strict

Temporal dead zone

GIT

What is git rebase?

How to handle conflicts?

What is git cherry pick?