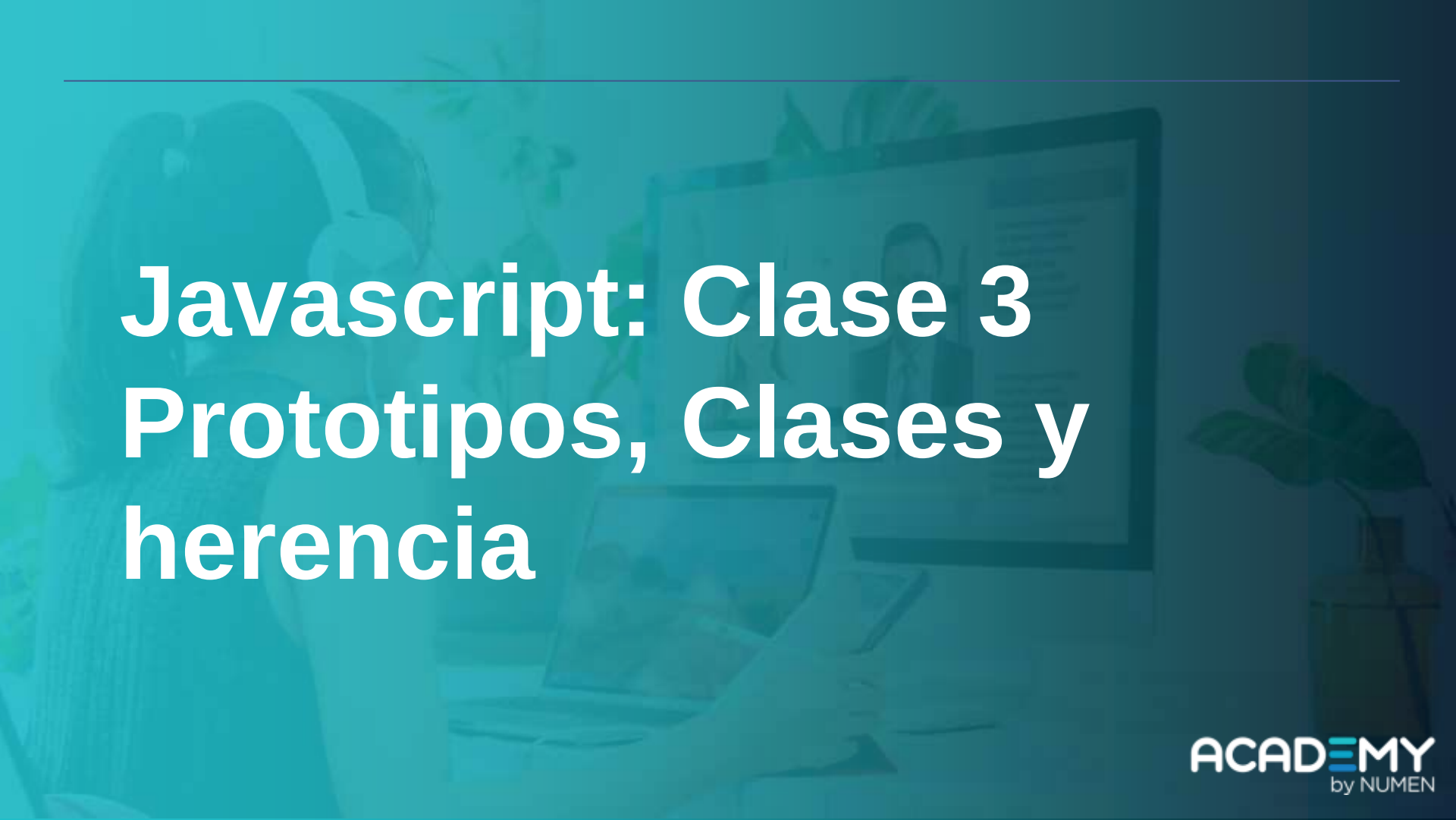
ACADEMY by NUMEN

FULL STACK

Comenzamos en unos minutos


ACADEMY
by NUMEN

Javascript: Clase 3 Prototipos, Clases y herencia



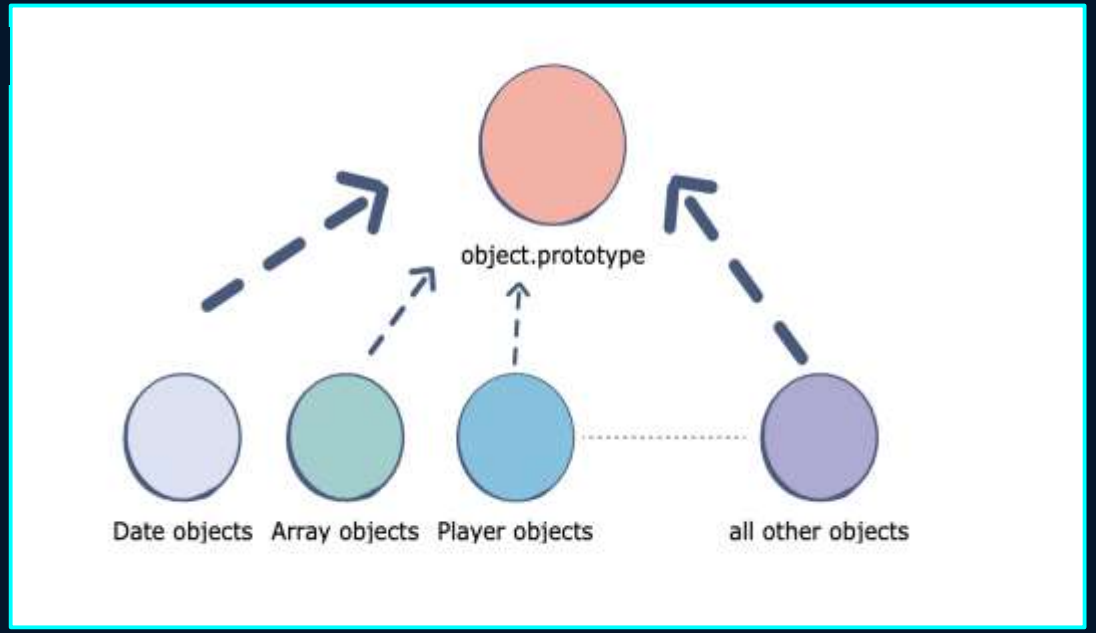
ACADEMY by NUMEN

¿Programación orientada a objetos (POO)?

Es un **modelo de programación** informática que organiza el diseño de software en torno a datos u objetos, en lugar de funciones y lógica. Se centra en los **objetos** que los desarrolladores quieren manipular en lugar de enfocarse en la **lógica** necesaria para manipularlos. Este enfoque de programación es adecuado para programas que son grandes, complejos y se actualizan o mantienen activamente.

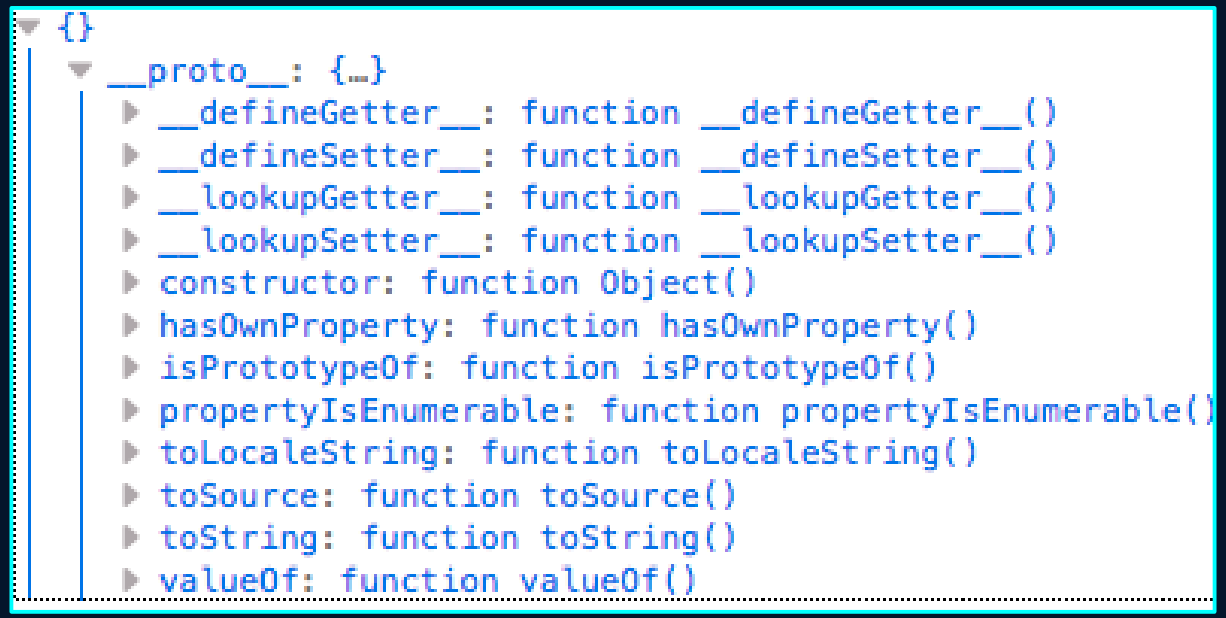
Javascript - Orientado a objetos basado en prototipos

JavaScript es a menudo descrito como un **lenguaje basado en prototipos** - para proporcionar mecanismos de herencia, los objetos pueden tener un **objeto prototipo**, el cual actúa como un objeto plantilla que hereda métodos y propiedades.



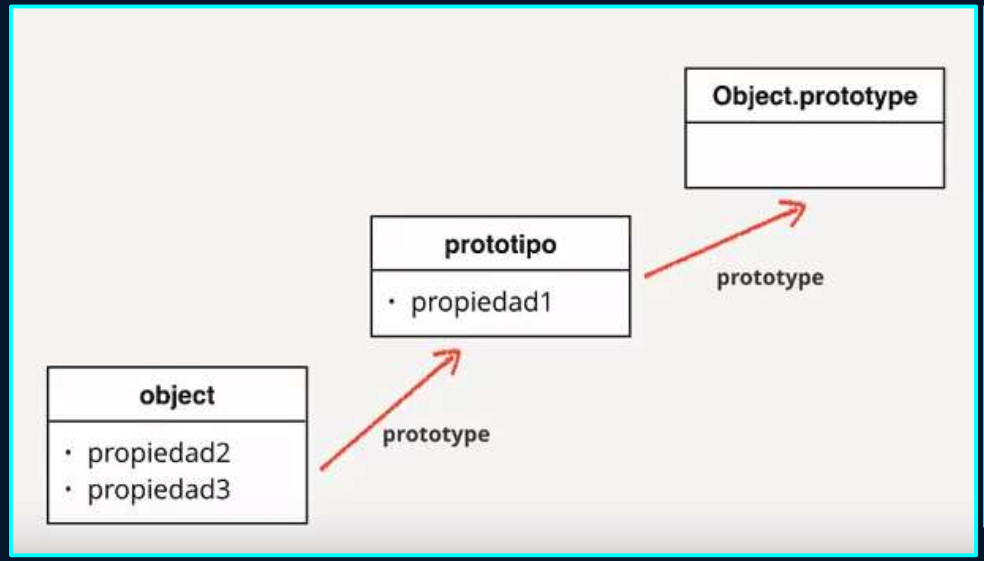
¿Qué es un prototipo (prototype)?

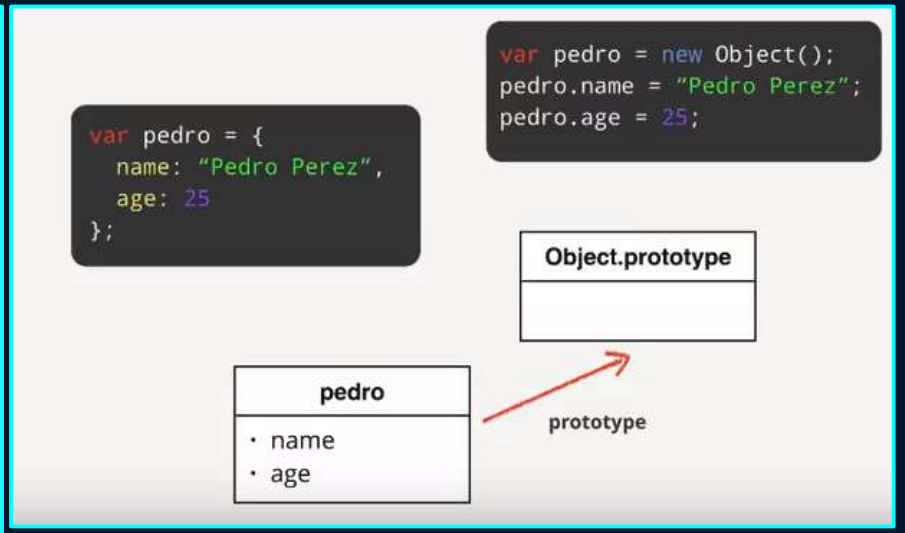
Un objeto en JavaScript tiene otro objeto, llamado ***prototype***. Cuando pedimos a un objeto una propiedad que no tiene, la busca en su prototipo. Así, un prototipo es otro objeto que se utiliza como una fuente de propiedades alternativa.



Herencia

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**.

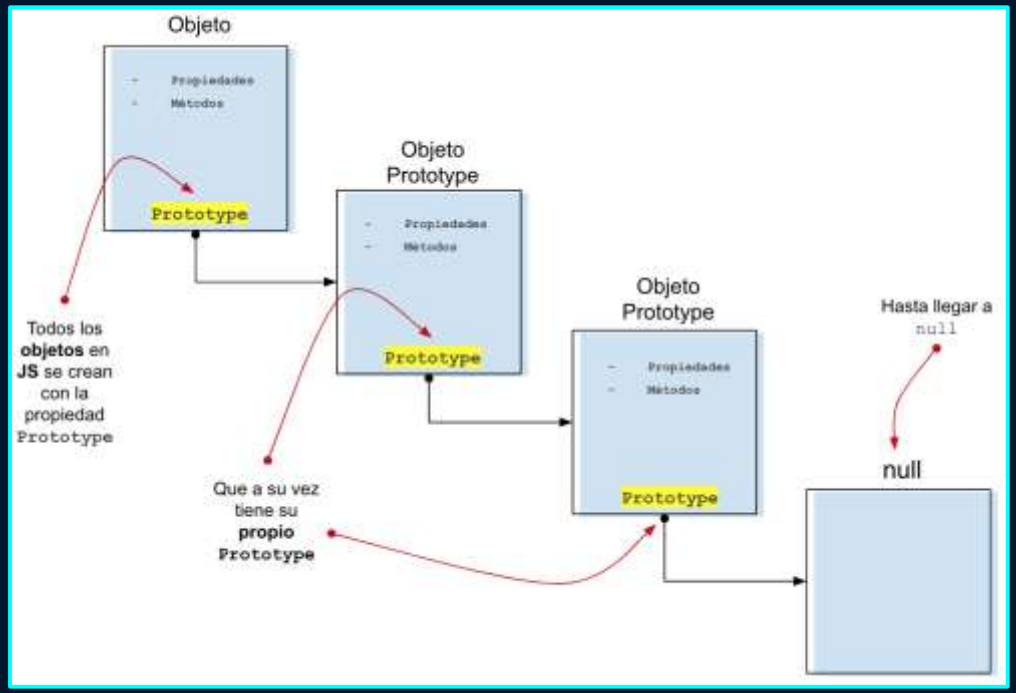




ACADEMY by NUMEN

Cadena de prototipos

Los objetos en JavaScript son "contenedores" dinámicos de propiedades (referidas como sus **propiedades particulares**). Los objetos en JavaScript poseen un enlace a un objeto prototipo. Cuando intentamos acceder a una propiedad de un objeto, la propiedad no sólo se busca en el propio objeto sino también en el prototipo del objeto, en el prototipo del prototipo, y así sucesivamente hasta que se encuentre una propiedad que coincida con el nombre o se alcance el final de la cadena de prototipos.



This

En la mayoría de los lenguajes de programación, This es una palabra reservada que hace referencia al objeto en el que estamos trabajando.

A través de esta palabra podemos invocar propiedades y métodos que tenga el objeto en cuestión.

En Javascript funciona algo diferente debido a que esta palabra también toma en cuenta los contextos (scope).

This en el scope global

Cuando imprimos en la consola la palabra reservada This suelta en el documento podremos observar que se imprime el objeto Windows.

El objeto windows es un objeto que contiene propiedades y métodos que afectan al navegador.

This suelto en el contexto global hace referencia al objeto windows.

ACADEMY by NUMEN

Agregar propiedades al objeto windows

Sin necesidad del uso de una variable, es posible agregar nuevas propiedades al objeto global windows.

Como podrán observar en el código, hemos agregado una nueva propiedad string.

```
// Agregando propiedad al objeto windows
this.nombre = "Bienvenidos al objeto windows"


console.log(this.nombre)
```

This en el scope de bloque

¿Qué pasaría si ahora intentamos imprimir esa propiedad desde dentro de, por ejemplo, un objeto?

Veamoslo!!

```
// Imprimiendo this.nombre desde un objeto
const objeto = {
    nombre: 'Contexto del Objeto',
    imprimir: function () {
        console.log(this.nombre)
    }
}

objeto.imprimir();
```

Llamando This desde otro objeto

¿Qué pasaría si ahora yo quiero llamar a la función imprimir de este objeto desde otro objeto?

¿Imprimiría su propio nombre o el nombre del objeto del cual estoy llamando la función?

Vamos a verlo!!

```
// Imprimiendo this.nombre desde un objeto
var objeto = {
    nombre: 'Contexto del Objeto',
    imprimir: function () {
        console.log(this.nombre)
    }
}

// Llamando la función imprimir desde otro objeto
var objeto2 = {
    nombre: 'Contexto del Objeto 2',
    imprimir: objeto.imprimir()
}

objeto2.imprimir()
```

Función constructora

La **función constructora** es la versión de JavaScript de una **clase**. Notarás que tiene todas las características que esperas en una función, aunque no devuelve nada o crea explícitamente un objeto — básicamente sólo define propiedades y métodos.
Para declarar una función constructora se crea la función igual que cualquier otra función y después se crea un objeto utilizando la palabra clave **new** lo cual indica que la función original es una función constructora.

```
// Función constructora
function Jedi() {
    this.arma = 'Sable de Luz';
    this.poder = 'La fuerza'
}

// Creación de instancias
let jedi = new Jedi();

// Imprimimos en la consola
console.log(j)
```

Recibiendo métodos

Las propiedades de una función constructora se denominan **Atributos**. Además de estos atributos, también pueden recibir métodos.

Para poder llamarlos necesitamos hacer referencia a la función, a su prototipo y al método en cuestión.

```
Jedi.prototype.susurrar = function () {
    console.log('Usa la fuerza Luke')
}
```

```
// Función constructora
function Jedi(arma, poder) {
    // Atributos
    this.arma = arma;
    this.poder = poder;

    // Metodos
    this.hablar = function () {
        console.log('Que la fuerza te acompañe')
    }
}

let yoda = new Jedi('Sable laser', 'La fuerza')

console.log(yoda)
```

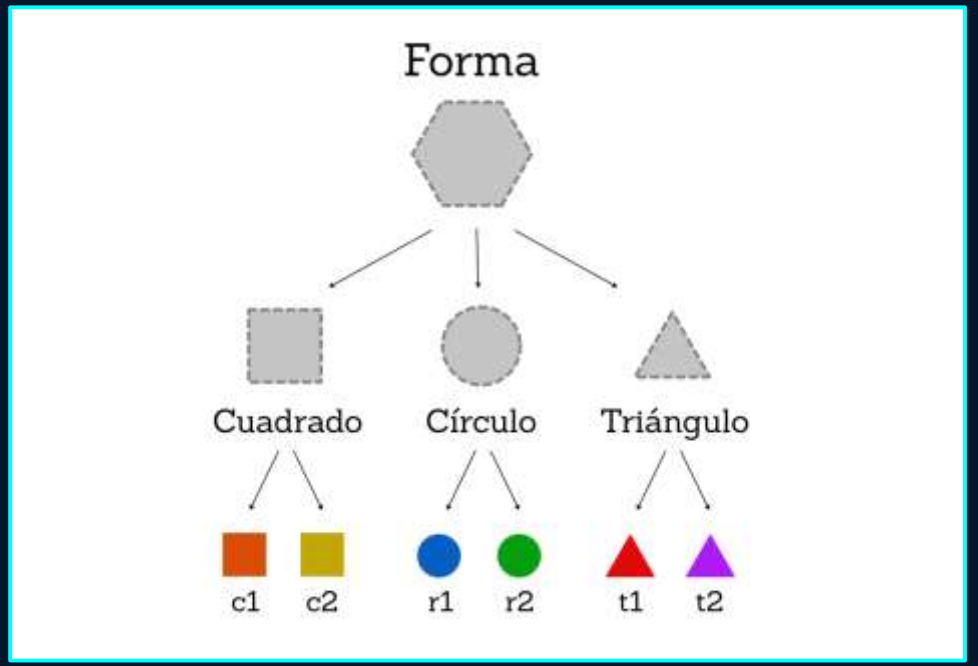
Clases

Las clases son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript. La sintaxis de las clases no introduce un nuevo modelo de herencia orientada a objetos. Proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia.
Una manera de definir una clase es mediante una declaración de clase.
Para declarar una clase, se utiliza la palabra reservada **class** y un **nombre**.

```
class Jedi {
    constructor (nombre, raza){
        this.nombre = nombre,
        this.raza = raza
    }

    saludar() {
        console.log('Hola! ' + this.nombre)
    }
}

let yoda = new Jedi('Yoda', 'Desconocida');

yoda.saludar();
```

Herencia de clase

Una de las características fundamentales que nos ayudan a reutilizar código y simplificar nuestro trabajo es la **herencia de Clases**.
Con esta característica podemos establecer una jerarquía de elementos y reutilizar características según en qué nivel se encuentra cada elemento.
Los elementos **hijos**, además de tener sus propias propiedades, pueden heredar las propiedades de su **padre**, y así sucesivamente.



Constructor

El método **constructor** es un método especial para crear e inicializar un objeto creado a partir de una clase.
Sólo puede haber un método especial con el nombre de "constructor" en una clase.
Un error de sintaxis será lanzado, si la clase contiene más de una ocurrencia de un método constructor.

Un constructor puede utilizar la palabra clave **super** para llamar al constructor de una clase padre.
Si no especifica un método constructor, se utiliza un constructor predeterminado.

```
class Jedi {

    constructor(nombre, alias, colorDeSable) {
        this.nombre = nombre;
        this.alias = alias;
        this.colorDeSable = colorDeSable
    }
}

class Yoda extends Jedi {
    constructor(nombre, alias, colorDeSable,
altura, vestimenta) {

        super(nombre, alias, colorDeSable)

        this.altura = altura;
        this.vestimenta = vestimenta
    }
}
```

ACADEMY
by NUMEN