Seminario JavaScript

Integrantes:

Richard Alejandro Matos Arderí

Abraham Romero Imbert

Mauricio Sunde Jiménez

Grupo 311, Ciencia de la Computación

Facultad de Matemática y Computación

Universidad de La Habana

2025

Introducción a JavaScript

Lenguaje de Alto Nivel

Interpretado y dinámico.

Desarrollo Web

Principalmente para interactuar con el DOM.

Características Clave

Interpretado, dinámico, orientado a objetos y eventos.

Modelo de objetos

JavaScript es un lenguaje de programación basado en prototipos, lo que significa que los objetos pueden heredar características de otros objetos directamente, a diferencia de los lenguajes basados en clases donde la herencia se realiza a través de las mismas. En JavaScript, cada objeto tiene un enlace a un objeto prototipo y puede heredar propiedades y métodos de él.

En este modelo los objetos no son creados mediante la instanciación de clases, sino mediante la clonación de otros objetos o mediante la escritura de código por parte del programador. De esta forma los objetos ya existentes pueden servir de prototipos para los que el programador necesite crear.

Cuando intentamos acceder a una propiedad de un objeto y no está presente en el propio objeto, JavaScript buscará esa propiedad en su prototipo. Si no la encuentra allí, buscará en el prototipo del prototipo, y así sucesivamente, hasta llegar al objeto Ob- ject.prototype, que es el último en la cadena de prototipos. Todos los objetos en JavaS- cript heredan propiedades y métodos de Object.prototype, lo que significa que métodos como toString(), valueOf(), hasOwnProperty(), entre otros, están disponibles para todos los objetos.

Cada función en JavaScript tiene una propiedad llamada prototype. Cuando se crea un objeto utilizando el operador new con una función constructora, el objeto recién creado hereda todas las propiedades y métodos definidos en la propiedad prototype de esa función constructora.

Modelo de Objetos en JavaScript

Basado en Prototipos

Objetos heredan directamente de otros objetos.

No usa clases tradicionales.

Cadena de Prototipos

Búsqueda de propiedades en el prototipo.

Hasta Object.prototype.

Funciones Constructoras

Crean objetos con el operador new.

Vinculan prototipos compartidos.

Creación de Objetos y Propiedades

Literales de Objeto

Sintaxis concisa para crear objetos.

```
let persona = {
nombre: 'Juan',
edad: 30,
saludar: function() {
console.log('Hola, soy ' + this.nombre);
} };
// Accediendo a las propiedades y metodos del objeto
console.log(persona.nombre);
persona.saludar();
```

Función Constructora

Uso de 'this' y 'new'.

```
function Persona(nombre, edad) { this.nombre = nombre; this.edad = edad; this.saludar = function() {
console.log('Hola, soy ' + this.nombre);
};
// Creando un objeto utilizando el constructor Persona
let persona1 = new Persona('Mario', 25);
```

Clase Object

Métodos asociados para creación.

```
let libro = new Object(); libro.titulo = 'El Aleph' libro.autor = 'Jorge Luis Borges' libro.mostrarInfo = function() {
console.log('Titulo: ' + this.titulo + ', Autor: ' + this.autor);
};
```

Prototipos y Herencia

Cada objeto tiene una referencia interna a otro objeto llamado su prototipo. La he- rencia en JavaScript se logra mediante la cadena de prototipos, donde un objeto puede acceder a las propiedades y métodos de su prototipo y así sucesivamente hasta que se alcanza un objeto cuyo prototipo es null, que es el final de la cadena de prototipos.

Funciones Constructoras

Las funciones constructoras son funciones especiales que se utilizan para crear instan- cias de objetos. Al utilizar la palabra clave new con una función constructora, se crea un nuevo objeto cuyo prototipo está vinculado a la función constructora. Esto permite la creación de múltiples instancias con propiedades y métodos compartidos a través del prototipo.

Métodos y Herencia Prototípica

Los métodos en JavaScript son funciones asociadas a un objeto que pueden ser invo- cadas para realizar acciones específicas. La herencia prototípica permite que los objetos hereden métodos de sus prototipos, lo que facilita la reutilización de código y la creación de jerarquías de objetos.

El modelo de objetos en JavaScript es único debido a su naturaleza basada en proto- tipos, lo que proporciona flexibilidad y poder para la creación y manipulación de objetos. Comprender este modelo es esencial para desarrollar aplicaciones robustas y eficientes en JavaScript.

Diseño de Jerarquía con Prototipos

Para ilustrar cómo se puede emular una jerarquía de clases en JavaScript a través de prototipos, consideremos un ejemplo sencillo con vehículos. Supongamos que queremos crear una jerarquía de vehículos donde Vehículo es el prototipo base y Automóvil y Camión son especializaciones de Vehículo.

1	**Vehículo** Prototipo base general.
2	**Automóvil** Especialización de Vehículo.
3	**Camión** Otra especialización.

A continuación, se muestra un ejemplo de herencia prototípica utilizando vehículos.

Prototipo Vehículo

```
function Vehiculo(tipo) {
    this.tipo = tipo;
}
Vehiculo.prototype.mover = function () {
    console.log("Moviendo un " + this.tipo);
};
```

Prototipo Automóvil

```
function Automovil(marca) {
    Vehiculo.call(this, 'automovil');
    this.marca = marca;
}
Automovil.prototype = Object.create(Vehiculo.prototype);
Automovil.prototype.constructor = Automovil;
Automovil.prototype.mostrarMarca = function () {
    console.log("Marca: " + this.marca);
};
```

Prototipo Camión

```
function Camion(cargaMaxima) {
Vehiculo.call(this, 'camion');
this.cargaMaxima = cargaMaxima;
}
Camion.prototype = Object.create(Vehiculo.prototype);
Camion.prototype.constructor = Camion;
Camion.prototype.mostrarCargaMaxima = function () {
console.log("Carga Máxima: " + this.cargaMaxima + "kg");
};
```

Otra manera de crear prototipos

```
var personaProto = {
saludar: function () {
console.log("Hola, mi nombre es " + this.nombre);
}
};
function Persona(nombre, edad) {
this.nombre = nombre; this.edad = edad;
}
Persona.prototype = personaProto;
//uso de prototipo
var persona1 = new Persona("Juan", 30);
persona1.saludar(); // Salida: Hola, mi nombre es Juan
```

POO en JavaScript

¿Es necesario tener clases para ser orientado a objetos?

No es indispensable que un lenguaje disponga de clases para ser considerado orientado a objetos. Lo fundamental en la Programación Orientada a Objetos (POO) son los objetos y los mecanismos que permiten:

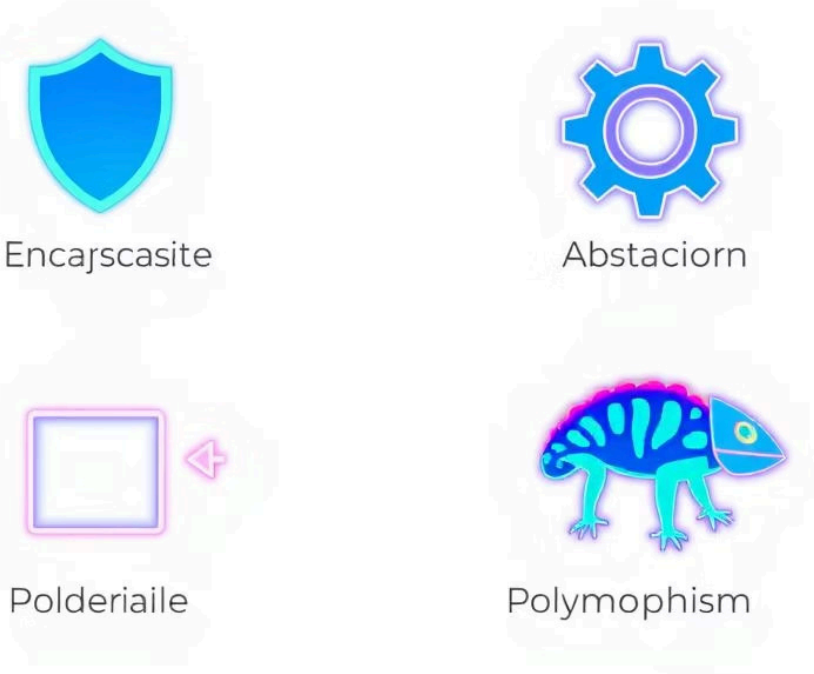
Encapsulamiento: Agrupar datos y comportamientos en una misma unidad.

Abstracción: Ocultar detalles internos y exponer una interfaz limpia.

Herencia: Compartir y reutilizar comportamiento entre objetos.

Polimorfismo: Permitir que objetos distintos respondan al mismo mensaje de formas diferentes.

JavaScript es un lenguaje orientado a objetos que, aunque basa su modelo en prototipos en lugar de clases tradicionales, cumple plenamente con los cuatro pilares fundamentales de la Programación Orientada a Objetos (POO):



POO en JavaScript: Pilares Fundamentales

Encapsulamiento

Agrupar datos y comportamientos.

Abstracción

Ocultar detalles internos.

Herencia

Reutilizar comportamiento.

Polimorfismo

Distintos objetos, mismo mensaje.

Encapsulamiento

El encapsulamiento agrupa datos y comportamientos en una única entidad: el objeto.

Ejemplo con objeto literal

```
const coche = {
  marca: "Toyota",
  encender: function() {
    console.log("Coche encendido");
  }
};

coche.encender()
```

Ejemplo con sintaxis de clases (ES6)

```
class Coche {
  constructor(marca) {
    this.marca = marca;
  }
  encender() {
    console.log(`${this.marca} encendido`);
  }
}
```

Abstraccion

La abstracción oculta la complejidad interna y muestra solo lo necesario.

Ejemplo de campos privados en ES2022 usando una clase:

```
class Cuenta {
  #saldo = 0; // propiedad privada

  depositar(cantidad) {
    if (cantidad > 0) this.#saldo += cantidad;
  }

  verSaldo() {
    return this.#saldo;
  }
}

const c = new Cuenta();
c.depositar(100);
console.log(c.verSaldo()); // 100
```

Herencia

JavaScript implementa herencia prototípica; desde ES6 también soporta sintaxis de clases.

Ejemplo de herencia prototípica:

```
function Animal(nombre) {
  this.nombre = nombre;
}
Animal.prototype.hablar = function() {
  console.log(this.nombre + " hace un sonido");
};

function Perro(nombre) {
  Animal.call(this, nombre);
}
Perro.prototype = Object.create(Animal.prototype);
Perro.prototype.constructor = Perro;
Perro.prototype.hablar = function() {
  console.log(this.nombre + " ladra");
};

const perro = new Perro("Firulais");
perro.hablar(); // Firulais ladra
```

Herencia con clases (ES6):

```
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }
  hablar() {
    console.log(`${this.nombre} hace un sonido`);
  }
}

class Perro extends Animal {
  hablar() {
    console.log(`${this.nombre} ladra`);
  }
}

const perro = new Perro("Firulais");
perro.hablar(); // Firulais ladra
```

Polimorfismo

El polimorfismo permite que distintos objetos respondan al mismo método de maneras diferentes.

```
class Animal {
  hablar() {
    console.log("Sonido genérico");
  }
}

class Gato extends Animal {
  hablar() {
    console.log("Miau");
  }
}

class Perro extends Animal {
  hablar() {
    console.log("Guau");
  }
}

const animales = [new Gato(), new Perro()]
animales.forEach(animal => animal.hablar());
```

Otras características relacionadas

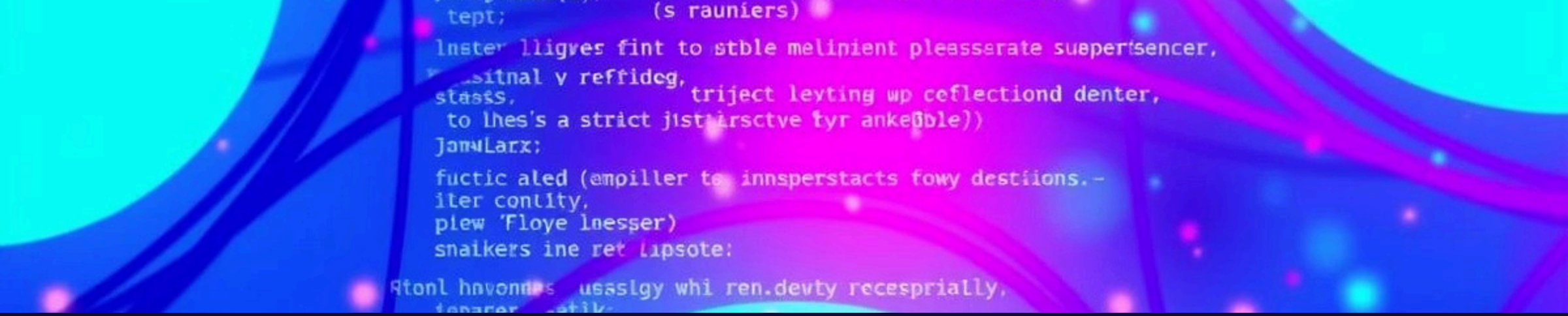
Funciones de primera clase: métodos son funciones que pueden pasarse como argumentos.

Objetos dinámicos: se pueden modificar (agregar/quitar propiedades) en tiempo de ejecución.

Closures: permiten emular encapsulamiento privado antes de los campos #private.

JavaScript cumple con los principios de la POO gracias a su modelo de objetos ba- sado en prototipos y, desde ES6, con sintaxis de clases y campos privados. Aunque no usara clases originalmente, incorpora todos los pilares fundamentales de la programación orientada a objetos.

Evolución de ECMAScript (ES6–ES14)



Conclusión: JavaScript y POO

1

Modelo de Prototipos

Flexibilidad y poder.

2

Sintaxis de Clases (ES6)

Facilita la POO.

3

Principios Fundamentales

Cumple con encapsulamiento, abstracción, herencia, polimorfismo.