13. Prototipos

Los **prototipos** son un concepto fundamental en JavaScript que permite la herencia y la reutilización de código. A diferencia de otros lenguajes que usan herencia clásica (como Java o C#), JavaScript utiliza un modelo basado en **prototipos**, lo que permite que los objetos hereden propiedades y métodos directamente de otros objetos.

A partir de ES6 se añadió una sintaxis más parecida a la de otros lenguajes de programación usando la palabra reservada `class` y refiriéndose a las superclases con `super`, pero no deja de ser lo que llaman en programación *"azúcar sintáctico"*. Una ayuda a la programación. Por debajo, el funcionamiento del código sigue basándose en los **prototipos**.

En este capítulo, exploraremos cómo funcionan los prototipos, su relación con las clases y cómo se implementan en JavaScript.

---

1. Clases vs. Objetos: Diferencias y Relación

Clases (ES6)

Las clases son plantillas para crear objetos. Introducidas en ES6, ofrecen una sintaxis más clara y estructurada para trabajar con objetos y herencia.

- Definen propiedades y métodos que se compartirán entre todas las instancias.
- Utilizan la palabra clave `class`.
- Los métodos definidos dentro de una clase se agregan automáticamente al prototipo del objeto.

Objetos

Instancias creadas a partir de una clase o función constructora.

- Contienen datos (propiedades) y comportamientos (métodos).
- Heredan propiedades y métodos de su prototipo.

Relación entre Clases y Objetos

- Las clases actúan como "fábricas" de objetos. Cada instancia de una clase es un objeto único que hereda propiedades y métodos del prototipo de la clase.

```
class Persona {
    constructor(nombre) {
        this.nombre = nombre;
    }
    saludar() {
        console.log(`Hola, soy ${this.nombre}`);
    }
}

const ana = new Persona("Ana"); // Instancia de Persona
ana.saludar(); // "Hola, soy Ana"
```

---

2. Función Constructora (Pre ES6)

Antes de ES6, las **funciones constructoras** eran la forma principal de crear objetos con propiedades y métodos compartidos. Aunque las clases son más modernas, las funciones constructoras siguen siendo relevantes para entender cómo funcionan los prototipos.

Ejemplo de Función Constructora:

```
function Persona(nombre) {
  this.nombre = nombre;
}

// Agregar un método al prototipo de la función constructora
Persona.prototype.saludar = function() {
  console.log(`Hola, soy ${this.nombre}`);
};

const luis = new Persona("Luis");
luis.saludar(); // "Hola, soy Luis"
```

Características Clave:

- `this`: Refiere al objeto que se está creando.
- `new`: Crea una nueva instancia y vincula `this` al nuevo objeto.
- `prototype`: Los métodos se agregan al prototipo para que todas las instancias los compartan.

---

3. Métodos Agregados al Prototipo

En JavaScript, los métodos definidos en el **prototipo** de una función constructora o clase son compartidos por todas las instancias. Esto optimiza el uso de memoria, ya que no se duplican en cada objeto.

Ejemplo con Función Constructora:

```
function Coche(marca) {
  this.marca = marca;
}

// Método agregado al prototipo
Coche.prototype.mostrarMarca = function() {
  console.log(`Marca: ${this.marca}`);
};

const toyota = new Coche("Toyota");
toyota.mostrarMarca(); // "Marca: Toyota"
```

Ejemplo con Clase (ES6):

```
class Coche {
  constructor(marca) {
    this.marca = marca;
  }

  // Método agregado al prototipo automáticamente
  mostrarMarca() {
    console.log(`Marca: ${this.marca}`);
  }
}

const tesla = new Coche("Tesla");
tesla.mostrarMarca(); // "Marca: Tesla"
```

4. La Cadena de Prototipos

Cada objeto en JavaScript tiene una propiedad interna llamada `[[Prototype]]`, que apunta a su prototipo. Cuando intentas acceder a una propiedad o método en un objeto, JavaScript busca en su prototipo si no lo encuentra en el objeto mismo. Esto se conoce como **cadena de prototipos**.

Ejemplo:

```
function Animal(nombre) {
  this.nombre = nombre;
}

Animal.prototype.sonido = function() {
  console.log("Sonido genérico");
};

function Perro(nombre) {
  Animal.call(this, nombre);
}

// Herencia: Perro hereda de Animal
Perro.prototype = Object.create(Animal.prototype);
Perro.prototype.constructor = Perro;

const max = new Perro("Max");
max.sonido(); // "Sonido genérico"
```

5. Buenas Prácticas

1. **Usar `class` para Sintaxis Clara**:
   Las clases simplifican la creación de objetos y la herencia, pero recuerda que internamente siguen usando prototipos.

2. **Evitar Modificar Prototipos Nativos**:
   Agregar métodos a prototipos como
   `Object` o `Array` puede causar conflictos con otras librerías o código futuro.
3. **Optimizar con Prototipos**:
   Los métodos compartidos deben definirse en el prototipo, no en el constructor, para evitar duplicar código en cada instancia.
