Universidad de La Habana

Facultad de Matemática y Computación

Lenguajes de Programación
Seminario 4 (JavaScript)

Juan Carlos Espinoza Delgado C-411
Raudel Alejandro Gómez Molina C-411
Alex Sierra Alcalá C-411

Proyecto en github

Introducción

Propósito del Surgimiento de JavaScript

JavaScript nació en 1995 de la mano de Brendan Eich en Netscape Communications, con el propósito de permitir a los desarrolladores agregar interactividad a las páginas web. A diferencia de lenguajes como Java, que requerían compilación y eran más complejos, JavaScript fue diseñado para ser sencillo, interpretado y embebido directamente en los navegadores. Su objetivo principal era facilitar la manipulación del Document Object Model (DOM) y permitir la creación de experiencias web más dinámicas y atractivas para los usuarios.

Problemas de Compatibilidad de los Navegadores Antes de la Adopción de ECMAScript 6

Antes de la estandarización y adopción de ECMAScript 6 (ES6) en 2015, el ecosistema de JavaScript enfrentaba serios desafíos de compatibilidad entre navegadores. Cada navegador (Internet Explorer, Netscape, Firefox, Chrome, Safari, etc.) implementaba el lenguaje de manera ligeramente diferente, lo que resultaba en:

- Inconsistencias en las API y Funcionalidades: Los desarrolladores tenían que escribir código específico para cada navegador o usar librerías como jQuery para abstraer estas diferencias.
- Problemas de Rendimiento: Diferentes motores de JavaScript (como V8 en Chrome, SpiderMonkey en Firefox) tenían variaciones significativas en cómo ejecutaban el código, lo que afectaba la experiencia del usuario.
- Falta de Características Modernas: Sin una actualización estandarizada, los desarrolladores no podían aprovechar características modernas y eficientes en todos los navegadores, limitando la capacidad de innovación y la complejidad de las aplicaciones web.

Actual Uso y Versatilidad de JavaScript

Desde la adopción de ES6 y posteriores actualizaciones de ECMAScript, JavaScript ha evolucionado enormemente, convirtiéndose en uno de los lenguajes de programación más versátiles y ampliamente utilizados en la industria del software. Actualmente, JavaScript es la columna vertebral del desarrollo web moderno, gracias a sus siguientes características:

- Compatibilidad y Estandarización: Con la adopción de ES6 y versiones posteriores, los navegadores modernos ofrecen una implementación más consistente y completa del lenguaje, reduciendo significativamente los problemas de compatibilidad.
- Versatilidad: JavaScript no solo se utiliza en el desarrollo frontend con tecnologías como React, Angular y Vue.js, sino también en el backend con Node.js, lo que permite a los desarrolladores usar un solo lenguaje en toda la stack de aplicaciones.
- Ecosistema Rico: Un vasto ecosistema de herramientas, librerías y frameworks facilita el desarrollo rápido y eficiente de aplicaciones web, móviles (React Native), de escritorio (Electron) e incluso en Internet de las Cosas (IoT).
- Innovación Continua: La comunidad de JavaScript y los comités de ECMAScript siguen introduciendo nuevas características y mejoras que mantienen al lenguaje relevante y poderoso para enfrentar los desafíos tecnológicos actuales.

En resumen, JavaScript ha recorrido un largo camino desde sus inicios como un simple lenguaje de scripting para navegadores, hasta convertirse en una pieza fundamental del desarrollo de software moderno. Su evolución ha permitido resolver problemas críticos de compatibilidad y ha potenciado su uso en múltiples contextos, demostrando una versatilidad y adaptabilidad que siguen siendo esenciales en la industria tecnológica actual.

Problemas Estructurales de JavaScript y el Surgimiento de TypeScript

Problemas Estructurales de JavaScript

A pesar de su popularidad y versatilidad, JavaScript presenta varios problemas estructurales que pueden complicar el desarrollo de aplicaciones complejas. Estos problemas incluyen:

1. Tipado Dinámico y Falta de Tipos Estáticos:
   - JavaScript es un lenguaje de tipado dinámico, lo que significa que las variables pueden cambiar de tipo durante la ejecución. Esto puede llevar a errores difíciles de detectar y depurar.
   - La falta de verificación de tipos en tiempo de compilación puede resultar en problemas de consistencia y errores de tipo que solo se manifiestan en tiempo de ejecución.
2. Problemas de Mantenimiento y Escalabilidad:
   - En proyectos grandes, la ausencia de un sistema de tipos robusto puede hacer que el código sea difícil de mantener y refactorizar.
   - La gestión de grandes bases de código puede volverse compleja debido a la falta de estructura y organización que un sistema de tipos podría proporcionar.
3. Herencia Prototípica:
   - Aunque JavaScript soporta herencia a través de prototipos, esta puede ser menos intuitiva y más propensa a errores que la herencia basada en clases, especialmente para desarrolladores provenientes de otros lenguajes orientados a objetos.
4. Gestión de Módulos:
   - Antes de ES6, JavaScript no tenía un sistema nativo de módulos, lo que dificultaba la organización y reutilización del código.
   - La gestión de dependencias y la modularización del código eran complejas y dependían de herramientas y patrones externos.
5. Problemas de Asincronía:
   - El manejo de operaciones asíncronas en JavaScript, tradicionalmente mediante callbacks, puede llevar al llamado "callback hell", haciendo que el código sea difícil de leer y mantener.

El Surgimiento de TypeScript

Para abordar muchos de estos problemas, Microsoft introdujo TypeScript en 2012. TypeScript es un superconjunto de JavaScript que agrega tipos estáticos y otras características avanzadas, con el objetivo de mejorar la productividad del desarrollo y la mantenibilidad del código.

1. Tipos Estáticos:
   - TypeScript introduce un sistema de tipos estáticos que permite a los desarrolladores definir y verificar los tipos de variables, funciones y objetos en tiempo de compilación.
   - Esto ayuda a identificar errores de tipo antes de que el código se ejecute, mejorando la confiabilidad y reduciendo el número de errores en tiempo de ejecución.
2. Compatibilidad con JavaScript:

- TypeScript es un superconjunto estricto de JavaScript, lo que significa que cualquier código JavaScript válido es también un código TypeScript válido.
- Esto permite a los desarrolladores adoptar TypeScript de manera incremental, comenzando con archivos JavaScript existentes y agregando gradualmente tipos y características de TypeScript.

3. Mejor Herramientas y Soporte para IDE:
   - TypeScript ofrece una integración superior con editores de código y entornos de desarrollo integrados (IDE), proporcionando autocompletado, refactorización y navegación mejorados.
   - Las herramientas de desarrollo se benefician del conocimiento del sistema de tipos, lo que facilita la escritura y mantenimiento del código.
4. Clases y Herencia:
   - TypeScript introduce una sintaxis de clases más familiar para los desarrolladores que provienen de otros lenguajes orientados a objetos, mejorando la legibilidad y el uso de patrones de diseño orientados a objetos.
   - Soporta herencia, encapsulación y polimorfismo de manera más intuitiva y estructurada.
5. Módulos y ES6+:
   - TypeScript soporta módulos ES6, permitiendo una mejor organización del código y gestión de dependencias.
   - Aprovecha las características modernas de JavaScript, asegurando que el código esté alineado con las mejores prácticas actuales.
6. Asincronía Mejorada:
   - TypeScript mejora el manejo de operaciones asíncronas mediante el uso de async/await, proporcionando una sintaxis más clara y directa para el manejo de promesas y funciones asíncronas.

TypeScript ha surgido como una poderosa herramienta para superar muchas de las limitaciones estructurales de JavaScript. Al agregar tipos estáticos, mejorar la organización del código y proporcionar una mejor experiencia de desarrollo, TypeScript ayuda a los desarrolladores a escribir código más robusto, mantenible y escalable. Esto ha llevado a su adopción creciente en la industria, especialmente para proyectos grandes y complejos donde las ventajas de un sistema de tipos sólido son más evidentes.

Modelo de objetos

JavaScript está diseñado sobre un paradigma simple basado en objetos. Un objeto es una colección de pares de ¡clave, valor¿ los cuales son llamados propiedades. El valor de una propiedad puede ser una función en cuyo caso se conoce como método.

Esta representación de los objetos al igual que en otros lenguajes de programación intenta simular el comportamiento de los objetos en la vida real.

Inicializadores de los objetos

Los inicializadores de objetos de se denominan también literales de objeto y utilizan la siguiente siguiente sintaxis:

```
const obj = {
  property1: value1, // property name may be an identifier
  2: value2, // or a number
  "property n": value3, // or a string
};
```

Las llaves de cada propiedad pueden ser cualquier expresión que pueda ser interpretada como un string, por ejemplo un número, un literal de string o una variable cuyo valor pueda ser interpretado como un string. Mientras que como valor de la propiedad puede aparecer cualquier expresión.

Los inicializadores de objetos son expresiones, y cada inicializador de objeto da como resultado la creación de un nuevo objeto cada vez que se ejecuta la instrucción en la que aparece. Los inicializadores de objetos idénticos crean objetos distintos que no se comparan entre sí como iguales.

Función constructora

Otra manera de inicializar los objetos es mediante el uso de una función constructora, mediante el uso de la de **this**.

Por ejemplo para crear un objeto **Car** con las propiedades **make**, **model** y **year** lo hacemos de la siguiente manera:

```
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
}
```

Luego podemos crear un nuevo objeto mediante el uso de **new**.

```
const myCar = new Car("Eagle", "Talon TSi", 1993);
```

Prototipos

En programación, la herencia se refiere a la transmisión de características de un padre a un hijo para que un nuevo fragmento de código pueda reutilizarse y basarse en las características de uno existente. JavaScript implementa la herencia mediante el uso de objetos. Cada objeto tiene un enlace interno a otro objeto llamado su prototipo. Ese objeto prototipo tiene un prototipo propio, y así sucesivamente hasta que se llega a un objeto con su prototipo. Por definición, no tiene prototipo y actúa como el eslabón final de esta cadena de prototipos. Es posible mutar cualquier miembro de la cadena de prototipos o incluso intercambiar el prototipo en tiempo de ejecución, por lo que conceptos como el envío estático no existen en JavaScript.

Los objetos JavaScript son "bolsas" dinámicas de propiedades (denominadas propiedades propias). Los objetos JavaScript tienen un vínculo a un objeto prototipo. Al intentar acceder a una propiedad de un objeto, la propiedad no solo se buscará en el objeto, sino también en el prototipo del objeto, el prototipo del prototipo, y así sucesivamente hasta que se encuentre una propiedad con un nombre coincidente o se alcance el final de la cadena del prototipo.

Para modificar o crear el prototipo de un objeto se puede hacer uso de la propiedad **__proto__**.

```
const o = {
  a: 1,
  b: 2,
  // __proto__ sets the [[Prototype]]. It's specified here
  // as another object literal.
  __proto__: {
    b: 3,
    c: 4,
  },
};

// o.[[Prototype]] has properties b and c.
// o.[[Prototype]].[[Prototype]] is Object.prototype (we will explain
// what that means later).
// Finally, o.[[Prototype]].[[Prototype]].[[Prototype]] is null.
// This is the end of the prototype chain, as null,
// by definition, has no [[Prototype]].
// Thus, the full prototype chain looks like:
// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> Object.prototype ---> null
```

```
20
21 console.log(o.a); // 1
22 // Is there an 'a' own property on o? Yes, and its value is 1.
23
24 console.log(o.b); // 2
25 // Is there a 'b' own property on o? Yes, and its value is 2.
26 // The prototype also has a 'b' property, but it's not visited.
27 // This is called Property Shadowing
28
29 console.log(o.c); // 4
30 // Is there a 'c' own property on o? No, check its prototype.
31 // Is there a 'c' own property on o.[[Prototype]]? Yes, its value is 4.
32
33 console.log(o.d); // undefined
34 // Is there a 'd' own property on o? No, check its prototype.
35 // Is there a 'd' own property on o.[[Prototype]]? No, check its prototype.
36 // o.[[Prototype]].[[Prototype]] is Object.prototype and
37 // there is no 'd' property by default, check its prototype.
38 // o.[[Prototype]].[[Prototype]].[[Prototype]] is null, stop searching,
39 // no property found, return undefined.
```

Herencia

Los objetos en JavaScript no poseen métodos al igual que los lenguajes de programación orientados a objetos, en cambio cualquier función puede ser atribuida a un objeto en forma de una propiedad. En consecuencia el comportamiento de un objeto padre puede ser heredado a un objeto hijo mediante una cadena de prototipos como vimos en la sección anterior.

```
1  const parent = {
2    value: 2,
3    method() {
4      return this.value + 1;
5    },
6  };
7
8  console.log(parent.method()); // 3
9  // When calling parent.method in this case, 'this' refers to parent
10
11 // child is an object that inherits from parent
12 const child = {
13   __proto__: parent,
14 };
15 console.log(child.method()); // 3
16 // When child.method is called, 'this' refers to child.
17 // So when child inherits the method of parent,
18 // The property 'value' is sought on child. However, since child
19 // doesn't have an own property called 'value', the property is
20 // found on the [[Prototype]], which is parent.value.
21
22 child.value = 4; // assign the value 4 to the property 'value' on child.
23 // This shadows the 'value' property on parent.
24 // The child object now looks like:
25 // { value: 4, __proto__: { value: 2, method: [Function] } }
26 console.log(child.method()); // 5
27 // Since child now has the 'value' property, 'this.value' means
28 // child.value instead
```

Incializar objetos usando las bondades de los prototipos

Supongamos que queremos crear varios objetos **Box** los cuales tendrán una propiedad **value** y un método **getValue**. Podríamos hacerlo de la siguiente manera:

```
const boxes = [
  { value: 1, getValue() { return this.value; } },
  { value: 2, getValue() { return this.value; } },
  { value: 3, getValue() { return this.value; } },
];
```

Esta implementación no es muy adecuada ya que estamos creando un nuevo espacio de memoria para cada instancia de **getValue**, en cambio podemos hacer que estos objetos compartan la implementación de **getValue** mediante el uso de un prototipo.

```
// A constructor function
function Box(value) {
  this.value = value;
}

// Properties all boxes created from the Box() constructor
// will have
Box.prototype.getValue = function () {
  return this.value;
};

const boxes = [new Box(1), new Box(2), new Box(3)];
```

De esta manera los 3 objetos compartirán la misma instancia de la función **getValue**.

Azúcar sintáctica para las funciones constructoras

Sin embargo el lenguaje nos proporciona una azúcar sintáctica para esta implementación mediante el uso de **class**:

```
class Box {
  constructor(value) {
    this.value = value;
  }

  // Methods are created on Box.prototype
  getValue() {
    return this.value;
  }
}
```

y ahora la herencia se puede implementar de la siguiente manera.

```
class Box {
  constructor(value) {
    this.value = value;
  }

  getValue() {
    return this.value;
  }
}

class ColoredBox extends Box {
  constructor(value, color) {
    // Llama al constructor de la clase padre
    super(value);
```

```
    this.color = color;
  }

  getColor() {
    return this.color;
  }

  getValue() {
    return `Value: ${this.value}, Color: ${this.color}`;
  }
}

// Ejemplo de uso
const myBox = new ColoredBox(10, 'red');
console.log(myBox.getValue());  // Output: Value: 10, Color: red
console.log(myBox.getColor());  // Output: red
```

Definiendo métodos get y set

Al igual que en los lenguajes de programación orientados a objetos en JavaScript es posible definir el método **get** y **set**, asignándole un método las llaves **get** y **set** del objeto respectivamente.

```
const myObj = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  },
};

console.log(myObj.a); // 7
console.log(myObj.b); // 8, returned from the get b() method
myObj.c = 50; // Calls the set c(x) method
console.log(myObj.a); // 25
```

Ventajas y desventajas del modelo de objetos basado en prototipos

Ventajas y Desventajas del Modelo de Objetos Basado en Prototipos

Ventajas

1. Flexibilidad y Dinamismo:
   - Modificación Dinámica: Los objetos pueden ser modificados en tiempo de ejecución, lo que permite cambiar su estructura y comportamiento dinámicamente.
   - Extensión de Objetos Existentes: Es fácil añadir propiedades y métodos a objetos ya existentes sin necesidad de definir nuevas clases.
2. Simplicidad Conceptual:
   - Sin Necesidad de Clases: No requiere la definición de clases; en cambio, los objetos pueden ser creados directamente y extendidos de manera simple.
   - Reutilización Directa: Permite la reutilización directa de objetos existentes como prototipos para nuevos objetos.

3. Herencia de Prototipos:

    - Cadena de Prototipos: La cadena de prototipos permite una forma sencilla y efectiva de herencia y reutilización de propiedades y métodos.
    - Mutación Prototípica: Se pueden cambiar los prototipos en tiempo de ejecución, permitiendo ajustar dinámicamente la herencia y el comportamiento de los objetos.

4. Simplicidad en el Envío de Mensajes:

    - Flexibilidad en el Envío: La resolución de métodos es dinámica y se realiza a través de la cadena de prototipos, lo que permite una mayor flexibilidad en la ejecución de métodos.

Desventajas

1. Complejidad en Grandes Proyectos:

    - Mantenimiento: En proyectos grandes, la falta de estructura de clases puede llevar a un código desorganizado y difícil de mantener.
    - Refactorización: La flexibilidad y dinamismo pueden dificultar la refactorización segura del código, especialmente en proyectos a gran escala.

2. Rendimiento:

    - Coste de Búsqueda en la Cadena de Prototipos: La búsqueda de propiedades y métodos a través de la cadena de prototipos puede introducir sobrecarga de rendimiento.
    - Mutaciones Dinámicas: Las modificaciones dinámicas en los prototipos pueden afectar negativamente al rendimiento del motor de JavaScript, que debe adaptarse a los cambios en tiempo de ejecución.

3. Inconsistencias y Errores:

    - Propiedades Heredadas: La herencia de propiedades y métodos a través de prototipos puede llevar a errores si no se manejan cuidadosamente las propiedades sobrescritas o heredadas.
    - Problemas de Contexto: El uso del contexto (this) en JavaScript puede ser confuso y propenso a errores, especialmente cuando se utilizan funciones como métodos.

4. Curva de Aprendizaje:

    - Concepto de Prototipos: Los desarrolladores acostumbrados a lenguajes basados en clases pueden encontrar confuso el modelo de prototipos y la herencia prototípica.
    - Contexto Dinámico: La dinámica del contexto y la necesidad de entender cómo funcionan los enlaces de prototipos pueden añadir complejidad para los nuevos desarrolladores.