Capítulo 1

Introducción a la tesis

1.1. Motivación

Hay una realidad de la que no se puede escapar, que es que hoy Javascript es uno de los lenguajes más utilizados a nivel empresarial. No solamente eso, sino que gracias a la aparición de Node en 2009, junto con una gran actualización del estandar ECMAScript en 2015 (también conocido como ES6), el lenguaje fue empezado a ser tratado de una forma más seria, abriéndose a nuevos dominios de aplicaciones, donde antes Javascript no era considerado como una posible solución.

Otro detalle, es que Javascript «ganó» demasiado terreno siendo el lenguaje pionero para ser implementado en los navegadores, además de que su comunidad ha crecido a un ritmo enorme en los últimos años. A nivel *frontend* en la web, al día de hoy los frameworks más populares usados son React, Angular y Vue, mientras que jQuery se quedó un poco atrás pero sigue en vigencia. Esos cuatro frameworks utilizan Javascript. Por el lado del *backend*, Node también está haciendo lo suyo. No solo eso, sino que además lo expande a otros dominios de aplicación, como pueden ser las aplicaciones de escritorio o la robótica.

Sin embargo es imposible evitar leer críticas del lenguaje, a veces infundadas. Agregando el dato de que la gama de opiniones y críticas es muy amplia, yendo desde el fanatismo del lenguaje hasta el odio profundo. Dicho esto, las motivaciones de hacer una tesis analizando el lenguaje está atada a los siguientes objetivos:

- Entender características profundas del lenguaje para comprender qué sucede a bajo nivel.
- Ganar experiencia en el área. Poder analizar, trazar y debuggear código con facilidad.
- Corroborar o desmitificar las críticas que se le suelen hacer al lenguaje.
- Analizar la relación del lenguaje con respecto a los paradigmas de programación populares.

1.2. Estructura de la tesis

El documento tiene un capítulo de introducción al lenguaje donde se presentará tanto su historia, como también características subyacentes y conceptos que serán

de uso para entender los capítulos siguientes. Luego, la tesis se divide en dos partes: «Sistema de tipos» y «Paradigmas de programación».

En la primera parte se hablará sobre características, debilidades y fortalezas dentro del sistema de tipos del lenguaje. Se hará mención sobre los tipos que tiene el lenguaje y algunas incoherencias sobre los mismos. También se revisarán cuestiones sobre coerción en algunas expresiones, y parte de cómo trabaja el scope del lenguaje.

La segunda parte trata sobre Javascript y los paradigmas de programación, donde se lo evalúa frente al paradigma funcional y el orientado a objetos. Se intentará determinar cuán cercano es el soporte que tiene el lenguaje ante las características de éstos paradigmas.

Cada parte tiene una sección de conclusiones, detallando *lo bueno, lo malo y lo feo* de los temas tratados. No se intenta imponer una opinión como una única verdad, sino más bien acercar al lector mostrando fortalezas y debilidades del lenguaje, para que éste pueda sacar sus propias conclusiones.

La tesis se resume con una conclusión general sobre lo visto, y una breve recopilación sobre conceptos que no han sido tratados en el documento, pero destacables en caso de que el lector quisiera seguir ampliando la investigaciónn.

Capítulo 2

Introducción a JavaScript

2.1. Historia de JavaScript

JavaScript, comunmente abreviado como JS, es un lenguaje de programación interpretado. En los comienzos, el lenguaje se utilizaba para agregar dinamismo del lado del cliente a las páginas web. Sin embargo, hoy en día se pueden crear aplicaciones de escritorio o del lado del servidor.

El lenguaje fue creado por **Brendan Eich** en 1995, quien en ese entonces trabajaba para Netscape. Eich denominó a su lenguaje `LiveScript`, y el objetivo inicial del lenguaje era solucionar problemas de validación de formularios complejos en el lado del cliente para el navegador Netscape Navigator, tratando de adaptarlo a tecnologías ya existentes.

La empresa Netscape junto con Sun Microsystems desarrollaron en conjunto este lenguaje de programación. Pero por cuestiones de mercado antes del lanzamiento, Netscape decidió cambiar el nombre del lenguaje a JavaScript (ya que en ese entonces Java estaba de moda en el mundo informático).

Al poco tiempo, la empresa Microsoft lanzó `JScript` para Internet Explorer. Para no entrar en una guerra informática, Netscape decidió que lo mejor sería estandarizar el lenguaje. Para ello, enviaron la especificación de JavaScript 1.1 al organismo ECMA (European Computer Manufacturers Association).

ECMA creó el comité TC39 con el objetivo de "*estandarizar de un lenguaje de script multiplataforma e independiente de cualquier empresa*". El primer estándar que creó el comité TC39 se denominó ECMA-262, en el que se definió por primera vez el lenguaje ECMAScript (abreviado comunmente como `ES`).

Es así entonces, que cuando hablamos de JavaScript, estamos haciendo referencia a una implementación de lo que se conoce como `ECMAScript`. El estándar ha ido evolucionando con el paso del tiempo. En la actualidad, la mayoría de los navegadores corren algún intérprete que soporta la mayoría de las características de las versiones 5.1 y 6.

Aunque la última versión sea la de ECMAScript 8 (lanzada en Junio de 2017), la versión 6 es más popular, ya que en ésta se han agregado muchos cambios significativos para el lenguaje. En este documento se hará énfasis en las versiones 5.1 y 6.

2.2. JavaScript en la actualidad

Después de más de 20 años de existencia, los usos del lenguaje han cambiado. JavaScript ya no es más un lenguaje para hacer validaciones de formularios complejos en páginas de Internet, ni tampoco para agregar dinamismo o animaciones a las páginas.

En la actualidad, se puede afirmar que JavaScript está en «la cresta de la ola». ¿Qué se puede hacer hoy en día con JavaScript?

- Páginas Web – Pareciera la respuesta obvia, sin embargo la forma de crear sitios web ha cambiado con el paso del tiempo. Hoy en día existe una gran cantidad de librerías y frameworks basados en JavaScript, tales como **React**, **AngularJS** o **Vue.JS**, entre otros.
- Aplicaciones móviles – Se pueden crear aplicaciones para celulares o dispositivos móviles programando en JavaScript, usando **Apache Cordova**, **Sencha**, **Ionic**, **NativeScript** o **Tabris.JS**.
- Aplicaciones de escritorio – Así como recién se hizo mención de las aplicaciones móviles, las de escritorio no se quedan atrás. Algunos frameworks como **Electron** ó **NW.JS** permiten crear aplicaciones multiplataforma.
- Robots – Mediante frameworks como **Cylon.JS** se pueden manejar dispositivos de hardware o robots. También existen kits basados en Arduino para programar en JS, tales como **Johnny-Five** o **Nodebots**.
- Aplicaciones de consola – Existen librerias que facilitan el uso de la creación de aplicaciones de linea de comandos (CLI).
- Machine Learning – Así como Python tiene una gran base de librerías para prototipar sistemas que apliquen Machine Learning, en este último tiempo la comunidad de JavaScript ha seguido los mismos pasos.

2.3. Características del lenguaje

JavaScript es un lenguaje de alto nivel, interpretado y multiparadigma. Es dinámica y débilmente tipado.

Posee herencia basada en prototipos. Este tipo de herencia es muy particular, y muy pocos lenguajes lo tienen.

Se dice que es multiparadigma porque soporta los paradigmas imperativo, funcional, orientado a objetos (prototipado) y dirigido por eventos.

En el ecosistema de la Web, JavaScript es uno de los lenguajes más populares. Todos los navegadores en la actualidad tienen un intérprete del lenguaje.

Si bien tiene bastantes partes criticables, JavaScript tiene la fama de ser un lenguaje «*liviano*» y «*expresivo*».

2.3.1. Influencias

JavaScript tiene fuertes influencias de varios lenguajes. Sus características más sobresalientes surgen de los siguientes lenguajes:

Java y C – No solo tiene la influencia sobre el nombre, sino que además tiene influencia sobre la sintaxis del lenguaje. Tanto Java como JavaScript sintácticamente emergen del lenguaje C. Sin embargo, Java y JavaScript tienen semánticas y propósitos diferentes.

Perl y Python – Tanto Perl como Python han influido en el manejo de strings, arreglos y expresiones regulares en JavaScript.

Scheme – De la familia del paradigma funcional. Adopta las funciones de primera clase y *closures*, los cuales se tratarán más adelante.

Self – Un lenguaje desarrollado por Sun Microsystems. Es de los pocos lenguajes que tienen herencia prototipada. Además de ésta característica, también se adopta la inusual notación de objetos.

2.3.2. Intérpretes

Ya se ha mencionado que JavaScript es un lenguaje interpretado. Sin embargo es necesario mencionar algunos «motores» que se encargan de interpretar el código en JavaScript.

Actualmente la gran mayoría de los navegadores (web browsers) viene con un intérprete de JS incorporado. A continuación se mencionan algunos de los más populares:

- **Rhino** – Gestionado por la fundación Mozilla, es de código abierto y está desarrollado completamente en Java.
- **SpiderMonkey** – También desarrollado por Mozilla para el navegador Firefox. Escrito en C++. Es utilizado en proyectos como MongoDB y GNOME.
- **Chakra** – Desarrollado por Microsoft, primero para Internet Explorer, y luego para Microsoft Edge.
- **V8** – El motor por defecto para Google Chrome, y también utilizado Node.JS, Opera y otros proyectos populares. Escrito en C++, maneja alocación en memoria y posee garbage collector.
- **JavaScriptCore** – Es utilizado por navegadores como Safari o PhantomJS. También es conocido como SquirrelFish o Nitro, bajo proyectos similares con otro nombre por cuestiones de mercado.

El objetivo de esta sección no es entrar en detalle ni hacer un análisis comparativo de los intérpretes. Basta con hacer una pequeña búsqueda para notar que varios de éstos intérpretes poseen garbage collection, compilación JIT (just in time), y estrategias para la optimización del código.

A lo largo de este documento se mostrarán ejemplos de código, cuya interpretación se realizará utilizando Node.JS (V8), y la consola de los navegadores Google Chrome (V8) y Mozilla Firefox (SpiderMonkey).

En caso de que el lector quiera ejecutar el código JavaScript, se deja a disposición los enlaces de descarga de las herramientas mencionadas:

- Node.JS – nodejs.org
- Google Chrome – google.com/chrome
- Mozilla Firefox – mozilla.org/firefox

Para abrir el intérprete desde Node.JS, basta con escribir `node` en la línea de comandos. Mientras que para el caso de los navegadores, hace falta apretar la tecla F12 para abrir la consola.

2.4. Nociones básicas

2.4.1. Tipos primitivos

Undefined

El tipo indefinido tiene un único valor, `undefined`. A toda variable que aún no se le haya asignado valor, tendrá el valor `undefined`.

Null

El tipo nulo tiene un único valor, `null`, que representa al valor nulo o «vacío».

Boolean

El tipo booleano representa una entidad lógica con dos posibles valores, `true` ó `false`.

String

Utilizado para representar datos de texto, el tipo String está definido como cero o más elementos, donde cada elemento es un entero no signado de 16 bits, de una longitud máxima de $2^{52} - 1$ elementos.

Number

Representa al conjunto de datos numérico. Se basa en la norma IEEE 754-2008, formato doble precisión de 64 bits en la aritmética de punto flotante. Toma algunos valores especiales de este conjunto para representar datos como NaN (Not a Number) y también +Infinity y -Infinity. La cantidad de valores reservados para NaN es dependiente de la implementación.

Symbol

Fue agregado en la versión de ES6. Abarca el conjunto de todos los valores no String que pueden ser usados como clave en la propiedad de un Object. Cada valor posible de Symbol es único e inmutable. Se los puede pensar como tokens que sirven como identificadores únicos.

Object

Es la forma básica de representar un objeto en JavaScript. Está compuesto por una colección de propiedades.

Las propiedades se identifican usando claves. El valor de una clave puede ser o bien un String, o bien un Symbol. Todos los valores String y Symbol son válidos como nombre clave para una propiedad, inclusive la cadena vacía.

2.4.2. Palabras reservadas

Las palabras reservadas del lenguaje se dividen en cuatro conjuntos:

- Palabras claves (*keywords*)
- Palabras reservadas a futuro
- Literal nulo (`null`)
- Literales booleanos (`true` y `false`)

Las siguientes son palabras claves, a excepción de `null`, `true` y `false`, que son literales.

CUADRO 2.1: Lista de palabras claves del lenguaje.

break	do	import	throw
case	else	in	true
catch	export	instanceof	try
class	extends	new	typeof
const	false	null	var
continue	finally	return	void
debugger	for	super	while
default	function	switch	with
delete	if	this	yield

Por otro lado, existe un conjunto de palabras reservadas a futuro. En un principio son solamente dos: `await` y `enum`. Pero si se especifica la directiva de *strict mode*, aparecen otras más: `implements`, `interface`, `package`, `private`, `protected` y `public`.

En resumen, las palabras reservadas a futuro (en modo estricto) son:

CUADRO 2.2: Lista de palabras reservadas a futuro.

await	implements	package	protected
enum	interface	private	public

2.4.3. Otras cuestiones a tener en cuenta

Identificadores

- Un identificador debe comenzar con una letra, signo pesos ($), ó guión bajo (_).
- Un identificador consiste en letras, números, signo pesos ($), ó guión bajo (_).
- Se permiten caracteres Unicode.
- No se permite el uso de palabras reservadas como identificadores.

Sensible a las mayúsculas

JavaScript es un lenguaje sensible a las mayúsculas, lo que significa que se entiende a `miVariable` y a `MIVARIABLE` como dos identificadores totalmente diferentes.

Sin tipado estático

El lenguaje no posee tipado estático. Sin embargo con **Microsoft TypeScript** o **Facebook Flow** se puede alcanzar esto mediante el uso de *type annotations*. Consiste en utilizar el lenguaje haciendo anotaciones de los tipos, para luego hacer un chequeo de tipos estáticos mediante un preprocesado del código. Tanto TypeScript como Flow son extensiones de JavaScript.

2.5. Sintaxis

A continuación se hará una introducción sintáctica al lenguaje de forma breve y mediante ejemplos. El objetivo de esta sección no es detallar la especificación del lenguaje, sino dar un repaso general por los elementos básicos, las estructuras de control y de repetición. Para mayor detalle sobre la sintaxis, se recomienda leer el «Standard ECMA-262 (Language Specification)».

2.5.1. Comentarios

Los comentarios en JavaScript se realizan de forma similar a los lenguajes influenciados por C (como por ejemplo JavaScript o C++). Es posible hacer comentarios inline, asi como también multilínea.

```
1 // Esto es un comentario en una sola linea
```

Comentario inline

```
1 /*
2 Esto es un comentario
3 escrito en varias lineas
4 */
```

Comentario multilinea

2.5.2. Variables

Para la declaración de variables, el lenguaje posee la palabra reservada `var`. Una variable tendrá el valor inicial `undefined` a menos que se la inicialice en su declaración.

También se pueden hacer múltiples declaraciones en la misma línea, incluso con la asignación de un valor inicial.

```
1 var a;                  // Definiendo una variable con nombre a
2 var b = 1;              // Definiendo una variable con nombre b
3 var c, d, e;            // Definiendo varias variables
4 var f, g = true, h;     // Esto tambien es valido
5 var i = "Hola", j = 2;  // Definiendo y asignando multiples variables
```

Declarando variables

Vale la pena hacer mención también a dos nuevas formas de definir variables a partir de ES6. Se trata de `let` y `const`.

Sobre `let`, es una forma de declarar variables de alcance local. Se hará énfasis en este punto en futuros capítulos cuando se muestren los problemas de alcance que posee el lenguaje.

Por el lado de `const`, se tratan de variables de valor constante, cuyo valor no se puede cambiar y tampoco pueden ser redeclaradas.

2.5.3. Estructuras condicionales

El lenguaje posee las estructuras condicionales ya conocidas en lenguajes como C++ o Java. A continuación se muestran algunos ejemplos de algunas, que servirán para mejor entendimiento del código de ejemplo dado en los otros capítulos.

Condicionales `if` e `if-else`

```
1 if (a > 0) {
2   // bloque si la condicion es verdadera
3 }
```

```
4
5  if (a > 0) {
6    // bloque si la condición es verdadera
7  } else {
8    // bloque si la condición es falsa
9  }
```

Ejemplos de `if` e `if-else`

Operador ternario `?:`

```
1  var mayor = a > b ? a : b
```

Operador ternario `?:`

2.5.4. Funciones

Las funciones en JavaScript son objetos, instancia de `Function`. Al ser objetos, las mismas pueden ser guardadas como valores dentro de variables. Existen diferentes maneras de declarar una función. A continuación se enumeran

Funciones como expresión

Una función como expresión (o función expresión) es una expresión que produce un valor, en este caso un objeto función, y luego es asignado a una variable.

```
1  var suma = function (x, y) { return x + y };
2
3  suma(2,3); // devuelve 5
```

Función expresión

Funciones como declaración

Una función como declaración (o función declaración) funciona de la misma forma que la función expresión, a diferencia de que no es necesario recurrir a la asignación sino que ésta se hace automáticamente.

```
1  function suma(x, y) { return x + y };
2
3  suma(2,3); // devuelve 5
```

Función declaración

Funciones anónimas y arrow functions

Se les llama funciones anónimas simplemente a aquellas funciones que no tienen nombre. Notar el detalle que cuando se habló de función como expresión, el lado derecho de la asignación era una función anónima.

Por otro lado, a partir de ES6 se introdujo el concepto de `arrow function` (o `fat arrow`, el cual está actualmente en varios lenguajes. Para el caso de JS, establece una forma más simple y concisa para escribir funciones anónimas, bajo el detalle de que además la palabra `this` dentro de la función no cambia ni está ligada a un nuevo contexto, sino que sigue está ligada léxicamente al contexto donde fue invocada. Esto último no sucede con las funciones como expresión comunes, ya que en éstas se crea un scope nuevo.

```
1 var mostrar = (texto) => console.log(texto);
2
3 mostrar('hola!'); // hola!
```

Arrow function

2.6. Otros conceptos

Para entendimiento del lector, se presentan algunos conceptos que se mencionarán en algunos capítulos. Algunos no son exclusivos del lenguaje, pero comprender y saber que el lenguaje dispone de éstas cualidades, ayudará a explicar cómo funcionan otras características que se presentarán más adelante.

2.6.1. Hoisting

Del inglés «levantamiento», el término de hoisting es muy acuñado y asociado a JavaScript. Consiste en mover hacia el comienzo del scope las declaraciones. En términos de compiladores, se puede pensar como una primera pasada del compilador para guardar los identificadores utilizados en el scope. De hecho, lo que sucede es que durante la fase de compilación se reservan en memoria espacio para los nombres declarados.

Gracias a ésta característica, se puede escribir invocar a una función cuyo código está más adelante.

```
1 foo()
2 function foo() {
3   console.log("bar");
4 }
```

En realidad lo que sucede es que el motor de JavaScript «mueve» las declaraciones al principio del scope, por lo que se puede pensar que el código del ejemplo es equivalente a éste:

```
function foo() {
  console.log("bar");
}
foo()
```

Para las variables sucede algo similar, aunque solamente en su declaración (no en su asignación), por lo que se puede pensar que el siguiente código:

```
console.log(a); // undefined
var a = 2;
```

Es equivalente a esto:

```
var a;
console.log(a); // undefined
a = 2;
```

La precaución hay que tenerla a la hora de escribir funciones como expresión, ya que el hoisting existe en la declaración del nombre pero no en la asignación (tal como pasa con las variables). Por ejemplo, el siguiente código:

```
foo();

var foo = function () {
  console.log("bar");
}
```

Nos lanzará en ejecución un `TypeError: foo is not a function`. Tiene sentido pensar en esto, ya que al igual que nos pasó con la variable `a`, el código equivalente sería:

```
var foo;

foo();  // en este punto, foo es undefined.

foo = function () {
  console.log("bar");
}
```

2.6.2. Closures

El término de closure, también conocido en español como «clausura» o «cerradura» es utilizado en otros lenguajes. Da la capacidad a las funciones a acceder y manipular variables que son externas a la función, siempre y cuando éstas estén definidas dentro del scope donde la función donde la función fue definida. Veamos un ejemplo simple:

```
1  var num = 2;
2  function imprimir() {
3    console.log(num);
4    num = 3;
5  }
6
7  imprimir();          // 2
8  console.log(num);    // 3
```

Creando un closure simple

Para este caso, la variable `num` y la función `imprimir` fueron definidas en el scope global. En la línea 3, la función puede entender al identificador `num` y ver su valor, e incluso en la línea 4, cambia su valor. Pero la fortaleza del closure no está ahí. Si nos limitásemos únicamente a hablar de visibilidad, deberíamos estar hablando de scope. La fortaleza del closure es que «encierra» a las variables que están dentro del scope de la función, y las mantendrá «vivas» siempre que la función exista. Veámoslo con un ejemplo más concreto:

```
1  function generar() {
2    var contador = 0;
3
4    function sumar() {
5      contador++;
6      console.log(contador);
7    }
8
9    return sumar;
10 }
11
12 var contar = generar();
13 contar(); // 1
14 contar(); // 2
15 contar(); // 3
```

Analizando otro closure

Se puede apreciar como la función `generar` es invocada por única vez. Al momento de invocar a la función, se crea un nuevo contexto. Sin embargo por causa del closure, ese contexto sigue con vida ya que es alcanzado desde el contexto global (mediante la variable `contar`) y así, el valor de `contador` persiste y cambia por cada llamada que hagamos a la funcion `contar`.

2.6.3. IIFE

La sigla IIFE representa *Immediately-invoked function expression*, lo que en español sería «función expresión invocada inmediatamente». El término es bastante autoexplicativo: Funciones definidas como expresión que son invocadas en el mismo lugar donde están definidas. Forman parte de un mecanismo importante que nos servirá más adelante para explicar los módulos en la sección 7.5.

Recordemos como mencionamos en la sección 2.5.4, existen dos formas de «crear» funciones: por declaración, o mediante una expresión. Las IIFEs corresponde únicamente a éstas últimas. Las dos maneras más comunes de escribir IIFEs se presentan a continuación.

```
// Dos versiones de IIFE. Ámbas válidas.
(function(){ /* codigo */ }());
(function(){ /* codigo */ })();

// Probándolas con console.log
(function(){ console.log("hola"); }()); // hola
(function(){ console.log("chau"); })(); // chau

// También pueden tener argumentos!
(function(msj){ console.log(msj); }("wow!")); // wow!
```

Introduciendo las IIFEs

¿Por qué se necesitan los paréntesis? Seguramente para que el intérprete entienda que se trata de una expresión y no de una sentencia. Sin embargo, éstas no son las únicas formas de generar IIFEs. A continuación se presentan más formas, aunque algunas de ellas un poco antiestéticas a la hora de analizar la legibilidad del código.

```
// Como una expresión del lado derecho de una asignación
var i = function(){ return 10; }();

// Dentro de una expresión booleana
true && function(){ /* codigo */ }();

// Con el operador coma
0, function(){ /* codigo */ }();

// Con operadores unarios
!function(){ /* codigo */ }();
~function(){ /* codigo */ }();
-function(){ /* codigo */ }();
+function(){ /* codigo */ }();

// Mediante el operador new, donde los paréntesis no son necesarios.
// Aún así, se los puede usar para pasar argumentos.
new function(){ /* code */ }
new function(){ /* code */ }()
```

Otras formas de escribir IIFEs

¿Qué ventajas nos brindan las IIFEs?

- Nos da la posibilidad de simular un scope local.
- Proveen un mecanismo de encapsulamiento.
- Reducen la polución de nombres en el scope global.
- Sirven para crear módulos o namespaces.

2.6.4. Prototype

Un concepto importante y clave para entender algunos temas a tratar más adelante es el de `prototype`. Recordemos que en JavaScript las funciones son objetos, y que los objetos son una colección de propiedades.

Al momento de declarar una función, se hace una ligadura de un identificador con el valor de un objeto que es instancia de `Function`. Dicho objeto tiene una propiedad especial llamada `prototype`, la cual es una referencia a otro objeto, inicialmente vacío. Análogamente, dicho objeto tiene una propiedad llamada `constructor` que hace referencia al objeto función que creó la instancia.

```
1  function Foo() {}
2
3  console.log(Foo.prototype); // {}
4
5  // agregamos propiedades al prototipo
6  Foo.prototype.valor = 42;
7  Foo.prototype.bar = function() {
8    console.log('bar');
9  };
10
11 console.log(Foo.prototype); // { valor: 42, bar: [Function] }
12
13 console.log(Foo.prototype.constructor); // [Function: Foo]
```

Analizando el `prototype` de una función

Podemos hacernos una imagen visual sobre las vinculaciones que hay en memoria al momento de la ejecución mediante el siguiente diagrama:

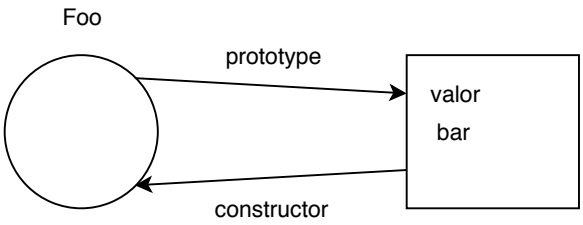

FIGURA 2.1: Diagrama del código y los objetos creados.

Ahora que se mencionó este concepto, tiene sentido revelar que el constructor de clase `Object` no es otra cosa más que una función, y sus métodos `toString`, `valueOf`, ó `hasOwnProperty` forman parte de su «prototipo». Ahora supongamos el siguiente código:

```
1  var a = new Object();
2
3  console.log(a.toString());
```

La instancia `a` también tiene una propiedad del prototipo. Pero en este caso, en vez de ser `prototype`, la misma se llama `__proto__`. Una definición vaga sería que las funciones están vinculadas a un prototipo mediante su propiedad `prototype` mientras que las instancias se vinculan a su prototipo mediante la propiedad `__proto__`. La forma «genérica» definida en el estandar de ECMAScript cuando se habla del prototipo, es mediante el término `[[Prototype]]`.

Ahora, ¿qué sucede exactamente al momento de hacer `a.toString()`? En realidad lo que sucede es que se busca dentro del objeto de la instancia de `a` por una propiedad

`toString`, pero al no encontrarse, se seguirá buscando en su `[[Prototype]]` (y en caso de no encontrarse, seguiría recursivamente hasta llegar a `null`). Ésto es lo que se denomina la cadena del prototipo («prototype chain»).

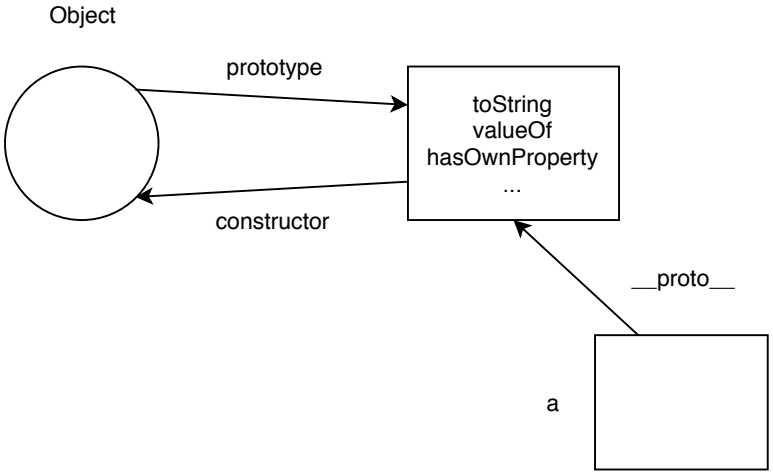

FIGURA 2.2: Diagrama del código y la instancia de `a`.

Sobre el uso del `prototype` se hablará con más detalle de esto en las secciones 7.1 y 7.2, donde en ésta última se hará mención a su rol en la herencia prototipada.

Parte I

Sistema de tipos

Capítulo 3

Sistema de Tipos

JavaScript es un lenguaje de «scripting», con tipado dinámico y un sistema de tipos débil. Generalmente, en este tipo de lenguajes interpretados no es necesario definir el tipo de una variable al momento de declararla, por lo que es lógico preguntarse ¿Es JavaScript un lenguaje seguro?. En este capítulo se abordará las cuestiones de JS relativas a su sistema de tipos.

3.1. Tipos primitivos

Tal como se mencionó en el Capítulo **??**, existen solamente 7 tipos primitivos del lenguaje. Una variable está asociada a un valor, y dicho valor puede ser de alguno de los siguientes tipos:

- `undefined`
- `null`
- `number`
- `string`
- `boolean`
- `symbol`
- `object`

Algunos autores consideran que `Object` no es un tipo primitivo, sino que es un tipo que hereda de `Null`. Por otro lado, hay que mencionar que las funciones en JavaScript son consideradas objetos, por lo que `Function` es un subtipo de `Object`.

Para el análisis del tipo de un valor (o del valor de una variable) en ejecución, se puede hacer uso del operador `typeof`. Dicho operador retorna un String con el nombre del tipo del valor evaluado.

```
1 typeof undefined     // retorna "undefined"
2 typeof 123           // retorna "number"
3 typeof true          // retorna "boolean"
4 typeof {}            // retorna "object"
5 typeof "Hola Mundo" // retorna "string"
6 typeof Symbol()      // retorna "symbol"
```

Analizando los tipos con `typeof`

Una de las primeras flaquezas presentadas por el lenguaje es la del `typeof null`.

```
1 typeof null          // retorna "object"
```

Analizando `typeof null`

Uno tiende a esperar que `typeof null` retorne "`null`", sin embargo, retorna "`object`". Este es un *bug* conocido y difícilmente sea corregido, ya que se estima que hay muchas aplicaciones y sistemas en la web que se basan en este comportamiento. Se cree que corregir esto crearía más problemas que soluciones.

Esto no solo pasa con el literal `null` sino que además sucederá con cualquier variable ligada a un tipo nulo.

```
1 var a = null;
2 typeof a           // retorna "object"
```

Analizando `typeof null` (cont.)

3.1.1. Casos especiales del `typeof`

Existen algunos casos especiales para el operador `typeof`. ¿Qué pasa con las funciones?. ¿Y con los arreglos?. ¿Y con los tipos built-in pero que no son primitivos (también conocido como «nativos»)?. Vamos por partes.

El primero de los casos es el de las **funciones**. Como se mencionó anteriormente, en la especificación, una función es considerada un subtipo de `object`, a diferencia de que tiene una propiedad interna `[[Call]]`. Sin ir más lejos, ¿qué se espera que retorne `typeof` para el caso de una función?

```
1 typeof function a(){}          // retorna "function"
```

Analizando `typeof` de una función

Si bien quizás resulte más intuitivo esperar que retorne "`object`", el hecho de que haya retornado "`function`"puede resultar útil a la hora de distinguir entre objetos y funciones, y así identificar cuales son los que se pueden invocar (también conocidos como «callable objects»). Sin embargo, este hecho es algo contradictorio ya que "`function`"no está distinguido entre los tipos primitivos.

El otro caso especial es el de los **arreglos**. En JavaScript, un arreglo no es más que un objeto con una propiedad interna `length`, donde cada propiedad de la instancia del objeto es el índice del arreglo.

```
1 typeof []                  // retorna "object"
2 typeof [1, 2, 3]           // retorna "object"
3 typeof ["hola", "mundo"]   // retorna "object"
```

Analizando `typeof` de arreglos

¿Por qué para las funciones el operador `typeof` devuelve `function` mientras que para los arreglos sigue devolviendo `object`?

Al parecer, la distinción de los «callable objects» es importante para el lenguaje, pero para el caso de arreglos es irrelevante saber si un arreglo es efectivamente un arreglo o simplemente un objeto, dado que tienen las mismas propiedades y se lo puede tratar de la misma manera. ¿Cómo saber entonces cuando -por ejemplo- una variable es un objeto o un arreglo? La respuesta es mediante el operador `instanceof`. La distinción se hace mediante el análisis de la clase asociada, y no del tipo.

3.2. Lo bueno, lo malo, lo feo...

Capítulo 4

Coerción

Cuando un lenguaje tiene seguridad en su sistema de tipos, hace que el mismo sea coherente, predecible, que sea fácil de intuir los tipos que se manejan. Para que un lenguaje gane flexibilidad en su sistema de tipos, debe sacrificar éstos aspectos de seguridad. Sin embargo, se pierden también los calificativos recién mencionados.

Una de las mayores críticas a Javascript es sobre la conversión de tipos implícita que ocurre en las expresiones. Tiene la fama de ser un lenguaje extremadamente flexible, pero al mismo tiempo impredecible en cuanto a la coerción.

En este capítulo se hablará de los efectos de la conversión de tipos que ocurre en el lenguaje, tanto la implícita como la explícita, y así podremos acercarnos a una conclusión de qué tan relajado es el sistema de tipos con respecto a la coerción. Para acotar el análisis, limitaremos el mismo a los operadores que se utilizan con más frecuencia a la hora de programar.

4.1. Conversión explícita

Comenzaremos este capítulo hablando de la conversión explícita, para entender qué valores de un tipo corresponden a otro tipo. Existen cuatro funciones para hacer conversión a tipos built-in del lenguaje, las cuales son las funciones `Boolean`, `Number`, `String` y `Object`. Es innecesario pensar funciones para convertir a `null` y `undefined` dado que son casos especiales.

4.1.1. Boolean

La función `Boolean()` convierte el argumento dado a un valor booleano. Los valores que se detallan a continuación se convierten a `false`, y son llamados valores de falsedad («*falsy*»).

- `undefined`
- `null`
- `false`
- `0`
- `NaN`
- `"`

El resto de los valores son considerados valores de verdad («*truthy*») y serán convertidos a `true`, incluyendo a los objetos, que todos son convertidos a true.

4.1.2. Number

Análogamente, la función `Number()` convierte un valor dado a su representación numérica. Para éste caso, las reglas son un poco más complejas (y poco intuitivas)

- `undefined` se transforma en `NaN`.
- `null` se transforma en `0`.
- `false` se transforma en `0`.
- `true` se transforma en `1`.
- Para el caso de `string`, se hace *parsing* de la siguiente manera: Se remueven los espacios en blanco, si el `string` resultante es `"`, se transforma en `0`, sino se intentará leer el valor numérico (si posee algún caracter ajeno a un valor numérico, se transformará en `NaN`, caso contrario, se transforma en dicho valor numérico).
- Para el caso de `object`, primero se transforma a primitivo (ver sección 4.2), y luego es convertido a `number` con las reglas recién mencionadas.

4.1.3. String

La función `String()` es el caso más obvio para la mayoría de los tipos primitivos. Convertirá el valor dado al literal string del tipo dado.

- `null` se convierte a `"null"`.
- `undefined` se convierte a `ündefined"`.
- `true` y `false` se convierten a `"true"` y `"false"`, respectivamente.
- Los valores numéricos se transforman en su `string` equivalente. Por ejemplo, `123.45` se transformará en `"123.45"`.
- Para el caso de `object`, primero se transforma a primitivo (ver sección 4.2), y luego es convertido a `string` con las reglas recién mencionadas.

4.1.4. Object

Para el caso de `Object()` es un poco más compejo de reflejar.

- `null` se convierte en el objeto vacío `{}`.
- `undefined` se convierte en el objeto vacío `{}`.
- Para los casos de `string`, `number` y `boolean`, las primitivas se convierten en «primitivas envueltas». Esto es, «objetos» que son instancias de `String`, `Number` y `Boolean`, respectivamente.
- Los objetos se vuelven en sí mismos.

4.2. ToPrimitive

La función `ToPrimitive` está especificada en el estandar de ECMAScript y es un pilar fundamental para entender las secciones siguientes. En la sección anterior (4.1) se hizo mención sobre la conversión de `object` hacia otros tipos. Para el caso de boolean, es simple dado que todos los objetos se convierten a `true`. Sin embargo, para los tipos `number` y `string` es un poco más complejo, dado que hay que recurrir a este algoritmo.

¿Por qué un objeto se querría convertir a un tipo primitivo? Es justamente una de las partes flexibles del lenguaje. Esta función es la que permite poder utilizar operandos de distintos tipos en algunas expresiones. Un objeto se podría tener que convertir a un tipo numérico para el caso de algunas operaciones matemáticas (por ejemplo, el tipo `Date` donde se pueden restar fechas) o por ejemplo a un texto, para el caso de que se necesite mostrar un objeto (ya sea mediante las funciones `alert` o `console.log`).

Cabe aclarar que la función `ToPrimitive` existe en la especificación de ECMAScript, pero no es un método o una función tal como se mostró con `Boolean`, `Number`, `String` y `Object`. Este método de conversión se utilizará siempre que el contexto lo requiera así, generalmente en operaciones donde ocurre la coerción.

La signatura del método es la siguiente: `ToPrimitive(input, PreferredType?)`

El parámetro opcional `PreferredType` representa la preferencia del tipo a convertir, y el mismo puede ser `Number` o `String`. Por lo general, se asume que cuando no está presente, el valor por defecto es `Number`, sin embargo este comportamiento puede ser cambiando sobreescribiendo el método `@@toPrimitive`, como sucede en el caso de `Date` y `Symbol`.

El algoritmo de `ToPrimitive` si el `PreferredType` es `Number` es el siguiente:

1. Si `input` es de tipo primitivo, retornarlo.
2. Sino, si `input` es un objeto, llamar a `input.valueOf()`. Si el resultado es primitivo, retornarlo.
3. Sino, llamar a `input.toString()`, si el resultado es primitivo, retornarlo.
4. Sino, lanzar un `TypeError`.

En caso de que el `PreferredType` sea tipo `String`, el algoritmo es análogo, solo que se cambian de lugares los puntos 2 y 3.

4.3. El operador +

Uno de los grandes dilemas a la hora de diseñar un lenguaje es sobre el uso de operadores como el +. ¿Debería este operador estar sobrecargado?. Es simple pensar en el + como un operador de suma de dos datos numéricos, pero ¿se podría usar para concatenar dos cadenas de texto? ¿o para unir el contenido de dos listas (arreglos)?.

4.3.1. Operador unario

La existencia del operador unario + puede resultar en una de las más confusas del lenguaje. La única finalidad del operador unario + es hacer una conversión numérica en caso de ser posible, y en caso contrario retornar `NaN`. No funciona de la misma forma en la que funcionaría el operador unario - (negación matemática), en donde además de realizar la conversión numérica, intentará cambiar el signo. Por ejemplo:

```
1  +3          // 3
2  -3          // -3
3  +(-3)       // -3
4  +-3         // -3
5  -+3         // -3
```

Operador unario +

Nuevamente, este operador funciona únicamente para una conversión a número, pero no del signo. Dicho esto, se presentan a continuación algunos ejemplos con sus respectivos resultados.

```
1  +3                                    // 3
2  +'-3'                                 // -3
3  +'3.14'                               // 3.14
4  +'3'                                  // 3
5  +'0xFF'                               // 255
6  +true                                 // 1
7  +'123e-5'                             // 0.00123
8  +false                                // 0
9  +null                                 // 0
10 +'Infinity'                           // Infinity
11 +'infinity'                           // NaN
12 +function(val){  return val }         // NaN
```

Operador unario + (más casos)

El operador unario + suele ser el elegido por algunos programadores cuando quieren hacer coerción explícita de un valor a su tipo numérico. Los resultados de utilizar `Number()` para ambos casos serían idénticos. El *trade-off* está en la legibilidad de las expresiones cuando éstas se vuelven más complejas.

4.3.2. Operador binario

Para comenzar, iremos con el ejemplo más simple de todos, el obvio: El operador + sirve como suma a la hora de trabajar con datos numéricos.

```
1  1 + 2        // 3
2  10 + 50      // 60
3  .1 + 0       // 0.1
4  NaN + NaN    // NaN
5  -1 + 10      // 9
```

Operador + en números

Por otro lado, el mismo operador también se puede usar para concatenación de strings.

```
"hola" + "mundo"  // "holamundo"
"abc" + "def"     // "abcdef"
"123" + "456"     // "123456"
"chau" + ""       // "chau"
```

Operador + en strings

Hasta este punto, todo parece funcionar de acuerdo a lo esperado. El problema es cuando empezamos a mezclar los tipos. Empecemos mirando los casos de `number` y `string`:

```
1 + "1"        // "11"
"1" + 1        // "11"
2 + ""         // "2"
"hola" + 0     // "hola0"
```

Operador + mezclando strings con números

Si bien puede parecer un poco raro, la regla aquí es que si uno de los operandos es `string`, entonces el resultado será la concatenación de los `string`. Ahora sigamos evaluando, qué pasa con los `boolean`:

```
true + true   // 2
true + false  // 1
false + false // 0
```

Operador + en booleanos

Tal como se mencionó en la sección 4.1, el valor numérico de `true` es `1` mientras que el de `false` es `0`. Entonces para este caso, ocurre una coerción hacia tipo numérico en los operandos, y el operador + funciona como una suma numérica normal.

Para entender mejor lo que está sucediendo, es preferible tener a mano un pseudo algoritmo de lo que sucede por detrás con el operador +:

1. Convertir ambos operandos a sus valores primitivos haciendo uso del `ToPrimitive` mencionado anteriormente.
2. Si alguno de los dos operandos es `string`, convertir ambos operandos a `string` y retornar la concatenación de los resultados.
3. Sino, convertir ambos operandos a su valor numérico y retornar la suma de los resultados.

Considerando esto, evaluemos algunos casos más:

```
[] + []      // ''
[] + {}      // '[object Object]'
{} + {}      // '[object Object][object Object]'
{} + []      // 0
```

Operador + con arreglos y objetos

Para el primer caso, la conversión a primitivo del arreglo vacío `[]` primero hace un `valueOf()`, el cual retorna el arreglo mismo (`this`). Luego, como el resultado no es un valor primitivo, se hace `toString()` el cual retorna el string vacío `''`. Finalmente, el resultado de sumar `[] + []` es el string vacío.

Para el segundo caso, sucede algo similar. Dado que el primer operando es el string vacío, entonces el + corresponde a una concatenación de strings. Y el método `toString()` sobre un objeto (instancia directa de Object) se traduce a string de la forma `'[object Object]'`.

Luego de haber visto los dos primeros casos, el tercer caso se explica solo. Se hace la conversión del primero y el + representa la concatenación de strings. Sucede exactamente lo mismo que en el primer caso, el `valueOf()` del primer operando retorna `this` entonces se termina haciendo un `toString()`.

El que es bastante inusual es el cuarto caso. Luego de haber visto el segundo, uno tiende a esperar que el cuarto de el mismo resultado, pero esto no es así. ¿Qué pasó? El intérprete consideró a la primera parte de la expresión como un bloque de código vacío y la ignoró. Luego, la expresión quedó como `+[]`, donde el + representa el operador unario, no el binario. Lo que significa que resultó en una coerción hacia `number` del string vacío `''`, resultando en un `0`. De hecho, podemos corroborarlo haciendo uso de los paréntesis, entonces el intérprete lo leerá como una expresión y `{}` representará un objeto en vez de un bloque.

```
1 ({} + [])     // '[object Object]'
```

Caso especial

Con todo esto visto, ahora se tiene un mejor panorama de cómo funciona el operador + en Javascript. Se utiliza principalmente en valores de tipos numéricos o strings.

Para el caso de los objetos, si alguno de los operandos es string, el mismo se convierte a string, en caso contrario, se convierte a número.

Para el caso de los arreglos, el operador + no funciona como en otros lenguajes. Para concatenar arreglos es necesario utilizar el método `concat` o hacer uso de alguna librería externa.

4.4. El operador !

El operador unario de negación lógica es otro de los operadores que es necesario hacer mención. Tal como sucedía con el operador unario +, en donde se hacía una conversión a Number, con el operador `!` sucede análogamente lo mismo, pero la conversión será a Boolean.

```
1 !true         // false
2 !false        // true
3 !null         // true
```

```
!undefined     // true
!0             // true
!1             // false
!{}            // false
![]            // false
!function(){} // false
```

Operador `!` con diferentes valores

Haciendo uso de este conocimiento sobre la conversión a `boolean`, podemos llegar a tener código más compacto. Por ejemplo, si tuvieramos una variable `a` y necesitamos chequear en algún momento si `a === null` o sí `a === undefined` para saber si `a` obtuvo algún valor, se puede hacer de la siguiente manera:

```
var a;
// ...
if (!a) {
  // lanzar un error
}
```

De todas formas, hay que tener especial cuidado con esto, ya que por ejemplo si nuestra variable `a` obtuviera el valor `0`, nuestro código lanzaría un error cuando en realidad sí se obtuvo un valor.

4.5. Operadores de igualdad

Una de las primeras fallas comunes en cuanto a los programadores que se acercan a Javascript y vienen de otros lenguajes de la rama de C, es pensar que el operador de comparación == funciona de la misma manera que en esos otros lenguajes. Esto no es así. En Javascript, existen dos operadores de comparación de igualdad, estos son == y === (junto con sus comparadores de desigualdad análogos, `!=` y `!==`).

El operador de igualdad === se le suele llamar estricto, ya que si los dos operandos son de distintos tipos, la expresión será `false`, en caso contrario, hará una comparación del valor de los operandos (a excepción de los objetos, que compara referencias).

En cambio el operador de igualdad == se le suele llamar blando, o según la especificación, «comparador de igualdad abstracta». Si los operandos son de distintos tipos, intentará convertir los tipos para luego compararlos.

Una de las mayores críticas del operador == es la falta de coherencia semántica. Por ejemplo, un arreglo vacío `[]` en el fondo es un objeto, lo cual debería ser un valor de verdad. La negación de un valor de verdad, por lógica, debería ser un valor de falsedad. Sin embargo, esto no sucede así.

```
[] == ![]          // true
```

Comparando un `Array` con su negación

¿Por qué ésta comparación da verdadera? Lo que está pasando en el fondo es que ambos operandos están siendo transformados a `number`. Por el lado de la izquierda, se produce el `ToNumber` visto anteriormente, resultando en un valor `0`. Sin embargo

el operando de la derecha previamente es transformado a `boolean` (por el `not`), lo cual se traduce a `false`, el cual luego termina siendo traducido a `0`.

```
+[] == +![];
0 == +false;
0 == 0;
true;
```

El caso de los arreglos no es lo único que carece de lógica. También hay un trato especial para `null` y `undefined`, los cuales se suponen que son valores de falsedad (tal como se marcó en 4.1), pero al momento de comparar los valores con `false` el resultado no es el esperado.

```
null == false       // false
undefined == false  // false
```

Comparando `null` y `undefined` con `false`

Para entender mejor qué pasa en el fondo, es necesario recurrir a los algoritmos de comparación de la especificación. Cabe la aclaración que se hará uso de tres «métodos»: `ToNumber` y `ToPrimitive`, los cuales se introdujeron en las secciones 4.1 y 4.2 respectivamente, y `Tipo` el cual retorna el tipo de un valor dado.

Empecemos con el algoritmo de **Comparación de Igualdad Estricta**. Sea la comparación `x === y`, donde `x` e `y` son los operandos, la comparación se hace de la siguiente forma.

1. Si `Tipo(x)` es distinto a `Tipo(y)`
2. Si `Tipo(x)` es Number, entonces
   a) Si `x` es NaN, retornar `false`
   b) Si `y` es NaN, retornar `false`
   c) Si `x` tiene el mismo valor numérico de `y`, retornar `true`
   d) Si `x` es +0 e `y` es -0, retornar `true`
   e) Si `x` es -0 e `y` es +0, retornar `true`
   f) Retornar `false`
3. Si `Tipo(x)` es Undefined, retornar `true`
4. Si `Tipo(x)` es Null, retornar `true`
5. Si `Tipo(x)` es String, entonces
   a) Si `x` e `y` tienen exactamente la misma secuencia de unidades de código (misma longitud y mismas unidades en los índices correspondientes), retornar `true`
   b) Sino, retornar `false`
6. Si `Tipo(x)` es Boolean, entonces
   a) Si ambos `x` e `y` son `true` o ambos son `false`, retornar `true`

   b) Sino, retornar `false`

7. Si `Tipo(x)` es Symbol, entonces

   a) Si ambos `x` e `y` tienen el mismo valor de Symbol, retornar `true`

   b) Sino, retornar `false`

8. Si `x` e `y` tienen el mismo valor de Object, retornar `true`

9. Sino, retornar `false`

Este algoritmo es dentro de todo bastante simple. Partiendo por el hecho de que si ambos tipos son distintos, se retorna false `false`, y sino, se hace una comparación por valores según el tipo. Cabe destacar que los casos de `NaN`, `+0` y `-0` son especiales para `number`, por su implementación de la norma IEEE 754. Matemáticamente, `+0` y `-0` son iguales, pero para la implementación, representan dos valores distintos, y por eso es que están ramificados de manera exclusiva en el algoritmo.

Ahora sigamos con el algoritmo de **Comparación de Igualdad Abstracta**.

1. Si `Tipo(x)` es igual a `Tipo(y)`, retornar el resultado de la comparación estricta `x === y`.

2. Si `x` es `null`, e `y` es `undefined`, retornar `true`

3. Si `x` es `undefined`, e `y` es `null`, retornar `true`

4. Si `Tipo(x)` es Number y `Tipo(y)` es String, retornar el resultado de `x == ToNumber(y)`

5. Si `Tipo(x)` es String y `Tipo(y)` es Number, retornar el resultado de `ToNumber(x) == y`

6. Si `Tipo(x)` es Boolean, retornar el resultado de `ToNumber(x) == y`

7. Si `Tipo(y)` es Boolean, retornar el resultado de `x == ToNumber(y)`

8. Si `Tipo(x)` es String, Number, o Symbol, y `Tipo(y)` es Object, retornar el resultado de `x == ToPrimitive(y)`

9. Si `Tipo(x)` es Object, y `Tipo(y)` es String, Number o Symbol, retornar el resultado de `ToPrimitive(x) == y`

10. Retornar `false`

Veamos el detalle, primero, que el algoritmo está definido de forma recursiva. Lo segundo a tener en cuenta, la mayoría de los tipos intentarán convertirse a Number a excepción de los casos del final donde incluye casos de Object los cuales son transformados con `ToPrimitive`. Sin embargo, previo a eso, hay casos especiales para `null` y `undefined`. Este algoritmo presentado es el que está en la versión de ECMAScript 6.0, sin embargo no se descarta que en versiones siguientes sea modificado.

La falta de «transitividad» en el operador == es otra de las cosas más preocupantes del lenguaje. Es fácil pensar que si `A == B` y que `B == C`, entonces `A == C`, pero esto no sucede siempre. Por ejemplo:

```
1 0 == ''             // true
2 0 == '0'            // true
3 '' == '0'           // false
4
```

```
5 false == undefined  // false
6 false == null       // false
7 null == undefined   // true
```

Falta de transitividad en ==

El consejo de los autores experimentados es siempre usar el operador ===, para ser estrictos con la comparación y no caer en transformaciones inesperadas. Sin embargo, a medida que mezclamos los operadores, podemos seguir cayendo en el mismo problema. Combinando lo visto en esta sección y en la sección 4.3 del operador +, tiene sentido entonces una expresión así:

```
1 true + true === 2   // true
```

Algo a tener en cuenta, y que no formará parte de este documento, es que este tipo de problemas no solo están presentes en la comparación de igualdad, sino que también existen dentro de otros operadores de comparación, como puede ser >, >=, <, <=, entre otros.

4.6. Aprovechando la coerción

Hasta este punto se ha hablado de los efectos inesperados o con poca coherencia que tiene la coerción en el lenguaje. Sin embargo, cuando uno conoce las reglas que tiene el lenguaje, puede empezar a hacer uso de esos efectos a su favor.

Recordemos principalmente que Javascript es, entre otras cosas, un lenguaje de «scripting», por lo que se espera que haya ciertas facilidades y no tanta burocracia delante del programador a la hora de realizar su tarea.

Si bien la coerción lleva a efectos inesperados, bugs y dificultad a la hora de trazar y encontrar errores en el código, es probable que los programadores experimentados destaquen el punto de que se pueden hacer cosas muy concisas.

Por ejemplo, supongamos un escenario de una llamada asincrónica donde nos puede venir `data`, asi como también puede faltar dicha información (que no nos llegue dicha propiedad), luego en el código podemos verificar que la misma «existe» mediante el uso de la coerción en una expresión.

```
1 if (data) {
2   // hacer operaciones
3 }
```

Otro ejemplo puede ser el de verificar si una función está definida o no en cuanto a su valor de verdad. Supongamos que tenemos tenemos una función que tiene un parámetro opcional `callback` y si ésta fue definida, queremos ejecutarla.

```
1 function procesar(callback) {
2   // hacer operaciones
3   callback && callback();
4 }
```

La línea 3 representaría un chequeo de «si callback fue definido, entonces ejecutarla». Si `callback` es `undefined` (porque se llamó a `procesar` sin argumentos), entonces la parte izquierda de esa expresión dará `false` y el operador `&&` hará un «cortocircuito», evitando ejecutar la segunda parte.

Estos dos ejemplos sirven para presentar al lector que, dentro del lenguaje, la coerción puede ser utilizada a favor del programador.

Capítulo 5

Scope

Otro de los puntos claves a analizar es el manejo de scope o ámbito.

5.1. Scope Léxico

Existen dos modelos predominantes cuando hablamos de scope: El léxico y el dinámico. Javascript hace uso del modelo de scope léxico, pero ¿qué significa exactamente esto?

El scope léxico, también llamado estático, hace la definición de los nombres durante la declaración, es decir, cuando estamos «escribiendo código».

5.2. Scope por Bloque

Uno de los mayores errores de los programadores que vienen de la rama de Java y C, es creer que Javascript por naturaleza implementa scope por bloques. ¿Qué significa tener scope en bloque? Supongamos el siguiente código

```
1 {
2   var a = "primera";
3   {
4     var a = "segunda";
5     console.log(a);
6   }
7   console.log(a);
8 }
```

Si existiera el scope por bloque en Javascript, entonces el código presentado debería imprimir «segunda» en la línea 5, y luego «primera» en la línea 7, dado que ésta última pertenece a otro bloque y hacemos referencia a otra variable `a`. Sin embargo, si ejecutamos el código, podemos observar que en ambas ocasiones imprimió «segunda».

Este tipo de scope se llama «por bloque» y existe en otros lenguajes como Java. Sin embargo, en Javascript no existe este concepto, o al menos no con la palabra `var` (más adelante en la sección 5.2.1, veremos cómo se introdujo en ES6).

Lo que realmente está sucediendo en el código anterior es que se está aplicando el concepto de *hoisting* sobre el código. En todo el bloque solamente existe una variable `a`. La segunda declaración de la variable es omitida, aunque no su asignación. Y como para crear un «nuevo scope» es necesario hacer una declaración de una función, entonces no existirán dos scopes, sino que solamente uno.

Supongamos este otro ejemplo:

```
1  function test(condicion) {
2    if (condicion) {
3      var a = "la condición era verdadera";
4      // ...
5    } else {
6      var b = "la condición era falsa";
7      // ...
8    }
9  }
```

En realidad, no sucede que `a` solamente será declarada si la condición dada es verdadera (ni tampoco `b` si la condición es falsa). Lo que se interpreta realmente con el código, es lo siguiente:

```
1  function test(condicion) {
2    var a, b;
3    if (condicion) {
4      a = "la condición era verdadera";
5      // ...
6    } else {
7      b = "la condición era falsa";
8      // ...
9    }
10 }
```

La declaración de las funciones se hacen al principio de la función, cuando «comienza» el scope. Éste es uno de los puntos claves del lenguaje para entender que, a pesar de la similitud sintáctica con otros lenguajes, la semántica no es la misma.

Volviendo al primer ejemplo entonces lo que sucede es lo siguiente:

```
1  {
2    var a;
3    a = "primera";
4    {
5      a = "segunda";
6      console.log(a);
7    }
8    console.log(a);
9  }
```

Las llaves `{}` de las líneas 4 y 7 son sintácticamente válidas, pero no producen nada nuevo adentro del código. Se pueden omitir y el significado del programa será el mismo.

A veces el scope por bloque es necesario para no contaminar el espacio de nombres.

```
1 for (var i=0; i<10; i++) {
2   console.log(i);
3 }
4
5 console.log(i);   // 10
```

Contaminando el espacio de nombres

En el código presentado, podemos decir que la variable `i` se mantiene «viva» dentro del scope del programa. De hecho, esto además de influir en la polución del espacio de nombres, podríamos decir que también tiene una implicancia con el *garbage collector*, dado que costará identificar qué variables tiene sentido «limpiar» y cuáles no.

Hasta estos últimos años, los programadores que requerían hacer uso de scope por bloques se veían forzados a usar técnicas o *hacks* para no caer en estos problemas. Algunas, por ejemplo, eran crear nuevos scopes declarando funciones anónimas o mediante el uso de IIFEs, o incluso hacer uso del `with` o del `try-catch` para simular un bloque. Por suerte, a continuación, veremos que esto fue «solucionado».

5.2.1. `let` y `const` en ES6

A partir del 2015, una característica incluída en el estandar de ES6 es el uso de las palabras reservadas `let` y `const`. Con la introducción de éstas, aparece entonces el sentido de scope por bloques dentro del lenguaje.

`let`

La palabra `let` se puede pensar de una manera similar a `var`. La diferencia entre éstas dos, es que la sentencia `var` declara variables dentro de un contexto de entorno variable, mientras que `let` lo hace en un entorno léxico. De ésta forma, alcanzamos lo que queríamos, que era el scope por bloques. Supongamos:

```
1 {
2   let a = "primera";
3   {
4     let a = "segunda";
5     console.log(a);
6   }
7   console.log(a);
8 }
```

En este caso, sucederá lo que no podíamos hacer en la sección 5.2. El programa mostrará por pantalla «segunda», correspondiente a la línea 5, y luego «primera», correspondiente a la línea 7.

Hay que tener especial cuidado con la palabra `let` por dos motivos. El primero de ellos, es que no existe el concepto de hoisting cuando usamos el `let`. La definición de la variable, y la ligadura con el nombre se hace de forma léxica, es decir exactamente en la línea donde fue declarada.

```
1 console.log(a);   // undefined
2
3 var a;
4
5 console.log(b);   // ReferenceError: b is not defined
6
7 let b;
```

Hoisting en `var` pero no en `let`

En el código presentado, en la línea 1 se imprime por pantalla `undefined` porque gracias al hoisting, esa declaración de la variable `a` sucede previo al primer `console.log`. Sin embargo no sucede lo mismo en la línea 5, dado que el identificador `b` aún no existe (será recién descubierto en la línea 7), entonces el programa lanza un `ReferenceError`.

Con lo otro que hay que tener cuidado, dado que ésta ligadura se hace de forma léxica, es que en los ciclos de repetición se hará tantas veces como se ejecute el ciclo. Por ejemplo:

```
1 for (let i=0; i<10; i++) {
2   console.log(i);
3 }
4
5 console.log(i); // ReferenceError
```

La ligadura de la variable `i` no solo se hace al comienzo del ciclo `for`, sino que además se hará en cada iteración.

`const`

Análogo a la palabra `let`, la palabra `const` también se usa para definir una variable en el código. La diferencia es que ésta se ligará a un valor constante, y cualquier intento de cambiar el valor de la misma resultará en un error.

```
1 const pi = 3.14;
2
3 pi = 3.1416;
```

Intentando cambiar el valor a una constante

El código recién presentado resultará en un `TypeError` en la línea 3, bajo la explicación de «Assignment to constant variable.» dado que se está haciendo una asignación a una constante. Sin embargo, hay que tener especial cuidado con ésta palabra, ya que solo funciona con tipos primitivos.

Para el caso de los objetos, el siguiente código no presentará ningún error:

```
1 const persona = { nombre: 'Juan' };
2
3 persona.edad = 25;
```

```
4
5 console.log(persona);   // { nombre: 'Juan', edad: 25 }
```

`const` sobre objetos

Para el caso de los arreglos, dado que son objetos, sucederá lo mismo:

```
1 const lista = [1, 2, 3];
2
3 lista.push(4);
4
5 console.log(lista);     // [1, 2, 3, 4]
```

`const` sobre arreglos

5.3. La palabra `this`

Si bien la palabra `this` no refiere directamente al scope, sino más bien a una cuestión de contexto, es necesario hacer un párrafo aparte sobre la misma, dado que el `this` puede cambiar de significado dentro del scope de una función.

Parte II

Paradigmas de programación

Capítulo 6

Paradigma funcional

Entre los puntos fuertes que posee el lenguaje, es indiscutible decir que las funciones son uno de los ejes principales. Sin embargo, ¿qué tan ligado está el lenguaje al paradigma de programación funcional?. ¿Soporta todas sus características?.

Es sabido que algunos conceptos como las funciones de alto orden están vinculadas con el paradigma funcional, mientras que también se conoce que JavaScript permite el pasaje de funciones como parámetros, o el retorno de las mismas como valores. En este capítulo se realizará un análisis sobre las mismas, para poder entender qué tan cerca o lejos está el lenguaje de las características del paradigma.

6.1. Recursividad

Al igual que los lenguajes sintáticamente similares a JavaScript, la recursión es una técnica que se aplica con naturaleza. Siempre que la función no sea anónima (es decir, que tenga un nombre), se puede aplicar recursión directa sin problemas. Además, el soporte para la recursión mutua también es posible gracias a la característica de *hoisting*.

Cualquiera de las formas vistas en el Capítulo 1 son válidas para definir una función recursiva. A continuación se presentan ejemplos de las mismas, mostrando recursión directa y cruzada:

```
1  // Función recursiva con función como declaración
2  function sumatoria(n) {
3    return n > 0 ? n + sumatoria(n - 1) : 0;
4  }
5
6  // Función recursiva con función como expresión
7  var factorial = function(n) {
8    return n > 0 ? n * factorial(n - 1) : 1;
9  };
10
11 // Recursión cruzada
12 var esPar = num => (num === 0 ? true : esImpar(num - 1));
13 var esImpar = num => (num === 0 ? false : esPar(num - 1));
```

Ejemplos de funciones recursivas

6.2. Funciones puras

Las funciones puras son otra de las claves del paradigma funcional. Esto es, funciones que bajo la misma entrada, siempre devuelven el mismo resultado. Esta cualidad brinda la propiedad de transparencia referencial entre los rasgos de un lenguaje perteneciente al paradigma funcional. Los beneficios que trae el uso de las funciones puras, son la testeabilidad, predictibilidad, y la falta de efectos colaterales.

Por suerte, en el lenguaje se pueden escribir fácilmente funciones puras. Lamentablemente el lenguaje no posee ninguna directiva, y tampoco existe una herramienta concreta para determinar la pureza (o impureza) de una función, más que el conocimiento del programador.

Algunos ejemplos de funciones puras se pueden ver a continuación:

```
1  function sumar(a, b) {
2    return a + b;
3  }
4
5  function esMayor(edad) {
6    return edad >= 18;
7  }
8
9  function calcularPrecio(cantidad, costo) {
10   return cantidad * costo;
11 }
```

Funciones puras

¿Por qué son funciones puras las presentadas previamente? Porque no causan efectos colaterales, y dependen únicamente de sus argumentos para generar un resultado. Ante los mismos argumentos, dichas funciones retornarán los mismos valores. Sin embargo, tal como mencionamos, es muy fácil caer en el uso de funciones impuras:

```
1  function obtenerDiaActual() {
2    return new Date().getDate();
3  }
4
5  function obtenerTextoPorId(id) {
6    return document.getElementById(id).textContent;
7  }
8
9  const PI = Math.PI;
10
11 function calcularArea(radio) {
12   return PI * radio * radio;
13 }
```

Funciones impuras

La función `obtenerDiaActual` retorna un valor «azaroso», y la misma llamada a la función puede retornar dos valores diferentes (dependiendo de la hora de la computadora). Para el segundo caso, `obtenerTextoPorId` depende necesariamente del estado de `document` que representa el DOM del navegador. Algo similar sucede con

`calcularArea`, incluso habiendo declarado a `PI` como una «constante», la función (de manera aislada) depende de un valor que está por fuera de su scope local, y no nos podemos asegurar de la ausencia de un efecto colateral.

Si bien el lenguaje no provee ningun mecanismo para identificar o limitar únicamente las funciones puras, a continuación se mencionan algunas prácticas para evitar las funciones impuras:

- Evitar el uso de variables que estén fuera del ámbito (scope) de la función. Esto incluye también a las variables globales del navegador.
- Evitar el uso del DOM o de variables del browser como `document` o `window`.
- Evitar el uso de `Math.random`, valores azarosos o cambiantes como el día y la hora actual.
- Evitar peticiones bajo el protocolo HTTP u otras que dependan del estado de la red.
- Evitar imprimir por pantalla o por consola, o cualquier mecanismo de I/O.
- Evitar la mutabilidad de datos. Recordar que los objetos en JS se manejan por referencia. Para éste ítem, se recomienda aplicar prácticas de inmutabilidad, ya sea mediante `Object.freeze` o el uso de alguna librería externa.

6.3. Funciones de primera clase y orden superior

Una de las características más destacables del lenguaje es que las funciones son objetos, que a su vez se pueden almacenar como valores. Gracias a esto, resulta extremadamente simple el pasaje de funciones como argumentos, así como también devolver funciones como un valor de retorno.

6.3.1. Funciones de primera clase

En JavaScript las funciones son objetos de primera clase, dado que las funciones son tratadas como cualquier otro valor. Al momento de crear una función, en realidad lo que se crea es un objeto de clase `Function`.

Para el caso de una función mediante declaración, se asignará a una variable ligada al nombre dado a la función (siempre que no sea anónima). Para el caso de la función por expresión, dependerá si la misma se realiza en una asignación.

```
1  function saludar() { console.log('hola'); }
2  var despedirse = () => console.log('chau');
3
4  console.log(saludar);     // [Function: saludar]
5  console.log(despedirse);  // [Function: despedirse]
6
7  console.log(saludar.toString());
8  // function saludar() { console.log('hola'); }
9  console.log(despedirse.toString());
10 // () => console.log('chau')
```

Analizando el valor de una función

El hecho de que las funciones sean tratadas como valores, a su vez, hace que se puedan crear objetos que dentro de sus pares clave-valor tengan propiedades que sean funciones. Es decir, métodos del objeto.

```
var gato = {
  nombre: 'Mishi',
  maullar: function() {
    console.log('miau');
  }
};

gato.maullar();
// miau
console.log(gato);
// { nombre: 'Mishi', maullar: [Function: maullar] }
```

Asignando una función como valor de una propiedad a un objeto

6.3.2. Funciones de orden superior

Dado que las funciones son tratadas como cualquier otro valor, el lenguaje permite la característica de que una función pueda ser pasada como argumento, o que sea el valor de retorno de otra función. Esto conlleva a alcanzar otros mecanismos, como es el curry en las funciones o la aplicación parcial.

Comencemos con algunos ejemplos:

```
var saludar = () => console.log('hola mundo!');

function ejecutar(fn) {
  fn();
}

function ejecutarSeguro(fn) {
  if (typeof fn === 'function') {
    fn();
  }
}

ejecutar(saludar);
ejecutarSeguro(saludar);
```

Pasando una función como argumento

En el código mostrado, se poseen dos funciones, `ejecutar` y `ejecutarSeguro`. La única diferencia es que éste último hace un chequeo de tipo del argumento brindado para asegurarse de que es una función y puede invocarla. Para el primer caso, si no le dieramos una función como parámetro, la ejecución de dicha llamada resultaría en un error.

La otra cualidad es la de retornar funciones, la cual se ejemplificará a continuación:

```
function generador(prefijo) {
  return texto => console.log(prefijo + ' ' + texto);
}

var imprimir = generador('Hola');

// En este momento, imprimir es una función!
console.log(typeof imprimir); // function

imprimir('mundo');    // Hola mundo
imprimir('che!');     // Hola che!
imprimir('a todos');  // Hola a todos
```

Retornando funciones

La función `generador` tiene un parámetro `prefijo` y retorna una función, que tiene un parámetro `texto`, y muestra mediante la consola la concatenación del prefijo y del texto. Dado que `generador` retorna una función, el valor de la misma es ligada a `imprimir`, por lo que tiene sentido pensar que éste es como un alias de la función y se puede invocar sin ningún inconveniente. En el ejemplo, se puede observar como `imprimir` de alguna forma «recuerda» el prefijo que fue establecido al momento de «generar la función».

Veamos otro ejemplo, haciendo una función suma que esté en una forma *curried*:

```
var sumar = a => b => a + b;

var sumar2 = sumar(2);
var sumar5 = sumar(5);

console.log(sumar2(10));  // 12
console.log(sumar5(10));  // 15
```

Función suma *curried*

Si bien la sintaxis de las *arrow functions* al principio puede resultar ajena, la función sumar es una función con un argumento `a` y retorna una función, que tiene un argumento `b`, la cual retorna la suma de `a + b`. De alguna forma, se puede observar como `sumar2` y `sumar5` son una aplicación parcial de la función `sumar`.

Otra forma de realizar aplicación parcián es mediante el método `bind` de la clase `Function`, el cual recibe como primer argumento el contexto (por lo general sería `this`), y luego una lista de argumentos, para retornar la función aplicada (total o parcialmente). Supongamos el siguiente código:

```
var multiplicar = (a, b) => a * b;

var duplicar = multiplicar.bind(null, 2);

console.log(duplicar(15));  // 30
```

Aplicación parcial usando `bind`

En este caso, `multiplicar.bind` nos retornará una función que será producto de aplicar parcialmente un 2 como primer argumento.

Las funciones de alto orden en el lenguaje son una de las características más poderosas que éste posee. La facilidad con la que se puede pasar o retornar una función brindan un mecanismo que otros lenguajes de programación no pueden alcanzar, y éste es uno de los puntos más fuertes de JavaScript.

6.4. Evaluación ansiosa y perezosa

Una característica deseable en los lenguajes del paradigma funcional es la de la evaluación perezosa (lazy). Sin embargo, la evaluación de expresiones en JavaScript se realiza de manera ansiosa (eager). Supongamos el siguiente código:

```
1 function elegir(condicion, opcion1, opcion2) {
2   if (condicion) {
3     console.log(opcion1);
4   } else {
5     console.log(opcion2);
6   }
7 }
```

Creando una función condicional

La función `elegir` es de alguna manera, la forma declarativa del `if-else`. Como primer parámetro recibe una condición, y como segundo y tercer parámetro recibe las opciones para el caso de que la condición sea verdadera o falsa.

Si el lenguaje implementara la evaluación perezosa, se espera que las expresiones no sean evaluadas hasta el momento de ser necesario (o postergar su ejecución al máximo). Es decir, si yo en `opcion1` u `opcion2` brindo como argumento una expresión, dicha expresión no debería ser evaluada si no entra por el flujo correspondiente del `if`. Si la condición es verdadera, `opcion2` no debe ser evaluada en ningún momento. Si la condición es falsa, entonces es `opcion1` la que no debe ser evaluada.

Por otro lado, aprovechando la libertad que tenemos en las expresiones gracias al sistema de tipos, podemos crear expresiones de la forma `true && console.log()`, o también una expresión matemática como `2 + console.log()`. Cualquiera de estas formas son válidas, más allá de la coerción que ocurra en dichas expresiones, al momento de analizar la expresión, se imprimirá el mensaje dado por pantalla.

Dicho esto, ejecutamos las siguientes invocaciones:

```
1 elegir(true, 'a', 'b' && console.log('Ansioso!'));
2 // Ansioso!
3 // a
4
5 elegir(false, 2 + console.log('Ansioso!'), 5);
6 // Ansioso!
7 // 5
```

Analizando resultados de las invocaciones

Como se puede analizar, para el primer caso, lo primero que ocurre es que se imprimime por pantalla el mensaje «Ansioso!», para luego imprimir «a». En el segundo

caso ocurre lo mismo: Primero se imprime «Ansioso!» y luego un «5». Con esto, se puede mostrar que el lenguaje hace la evaluación de las expresiones al momento de hacer la invocación, y no al momento de que sea realmente necesario evaluar la expresión. En resumen, JavaScript posee evaluación ansiosa.

6.5. Semántica de valores

Por lo general, en los programas del paradigma funcional no se realizan asignaciones, y así, el valor asoaciado a una variable nunca cambia. Este concepto está directamente relacionado con el de funciones puras y transparencia referencial: Al no tener una dependencia sobre el valor de una variable, se espera que una función se evalúe siempre de la misma forma para una misma entrada.

Esta característica de los lenguajes del paradigma funcional da un enfoque puramente matemático, eliminando la noción de estado. Sin embargo esto no sucede en JavaScript. Una variable puede cambiar su valor. En el lenguaje existen sentencias, entre las cuales una de ellas es la asignación. Bajo esta premisa, entonces tendremos una noción de estado en las aplicaciones de nuestro lenguaje, por lo que podemos decir que JavaScript no cumple con ésta propiedad.

Dado el siguiente código:

```
1  function generarContador() {
2    var contador = 0;
3    return {
4      contar: function() {
5        contador++;
6      },
7      mostrarContador: function() {
8        console.log(contador);
9      }
10   }
11 }
12
13 var obj = generarContador();
14
15 obj.contar();
16 obj.contar();
17 obj.mostrarContador();
```

Noción de estado

Podemos observar un ejemplo de lo fácil que es poseer la noción del estado. Para el caso, el objeto tiene una variable interna `contador` el cual puede ir cambiando a medida que se vaya invocando el método `contar` sucesivas veces. En éste sentido, no se cumple con la semántica de valores.

6.6. Conclusiones

- Lo bueno: JavaScript no será un lenguaje de programación funcional pura, pero el hecho de que las funciones sean valores dentro del lenguaje lo hace excesivamente poderoso. Poseer funciones como valores, permitir el paso de las

mismas como argumentos o como valores de retorno hace al lenguaje sumamente expresivo. Para la enseñanza de conceptos básicos de la programación funcional, con una sintaxis similar a la familia de lenguajes de C, es un buen lenguaje, más allá de la falta de todas las propiedades que corresponden a un lenguaje 100 % del paradigma funcional.

- Lo malo: La falta de evaluación perezosa, semántica de valores, transparencia referencial, entre otras cosas, lo alejan del paradigma funcional. Sin embargo, si tuviera éstas características seguramente tendríamos que limitar al lenguaje en otros aspectos, y probablemente quitarle el soporte para otros paradigmas de programación.

Capítulo 7

Paradigma orientado a objetos

Uno de los debates principales sobre JavaScript es su soporte hacia el paradigma orientado a objetos. Existe una amplia gama de opiniones y posturas con respecto a JavaScript y su uso dentro de dicho paradigma. Las opiniones pueden ir desde la negativa, es decir, que no lo soporta, hasta la positiva, pasando por un intermedio de que tiene cierto soporte, pero no naturalmente.

En éste capítulo se buscará realizar un análisis sobre características típicas del paradigma orientado a objetos, tratándo de ver de qué manera llega JavaScript a éstas características.

7.1. Clases

Previo a la salida de ES6 (es decir, en la version 5 de JavaScript) la creación de clases en JavaScript se realiza mediante el uso de patrones para la creación de objetos. La realidad es que en JavaScript no existe un soporte formal o natural para las clases, sino que hay que recurrir a estos patrones para simularlo. Los más populares, a mi entender, son:

- Factory class pattern
- Functional class pattern
- Prototype class pattern

7.1.1. Factory class pattern

Se trata de una función de tipo *factory* (fábrica) utilizada para crear elementos u objetos, con ciertas propiedades y bajo cierto comportamiento. En nuestro caso, dicha función será quien cree nuestras nuevas instancias de lo que queremos moldear como clase.

```
1 function animalFactory(nombre) {
2   var temporal = {};
3   temporal.nombre = nombre;
4   temporal.saludar = function() {
5     console.log("Hola, soy "+this.nombre)
6   };
7
8   return temporal;
```

```
}

var gato = animalFactory("Garfield");
var perro = animalFactory("Oddie");

gato.saludar();   // Hola, soy Garfield
perro.saludar();  // Hola, soy Oddie
```

Factory class pattern

Si bien puede resultar un poco confuso al principio, para quienes estén acostumbrados al patrón Factory, este ejemplo quizás resulte más trivial. A simple vista, la única *ventaja* es que no se necesita usar `new` a la hora de realizar la instanciación de nuevos objetos.

Para el objetivo que buscamos, que es simular el soporte de clases, este patrón es útil.

7.1.2. Functional class pattern

Este patrón también es conocido como Constructor pattern. Aprovechando el uso de la palabra `new`, podemos omitir la creación de un objeto temporal dentro de nuestra función. De hecho, la palabra `new` no solamente crea un nuevo objeto (instancia), sino que además establece quién fue la función de construcción (se puede pensar como «de quién hereda el objeto»).

```
function Animal(nombre) {
  this.nombre = nombre;
  this.saludar = function() {
    console.log('Hola, soy ' + this.nombre);
  };
};

var gato = new Animal('Garfield');
var perro = new Animal('Oddie');

gato.saludar();   // Hola, soy Garfield
perro.saludar();  // Hola, soy Oddie
```

Functional class pattern

Notar las diferencias con el ejemplo del Factory Pattern. Por un lado, el uso del `this` dentro de la función y la falta de necesidad de retornar el objeto (esto sucede implícitamente gracias al `new`). Por otro lado, a la hora de crear instancias es importante utilizar la palabra `new`.

Un detalle muy importante a tener en cuenta tanto en éste patrón como en el Factory pattern: Para cada instancia creada, la misma poseerá una copia del código de `saludar()` en memoria. Parece un detalle menor, pero supongamos que en vez de un solo método, nuestra clase tiene varios, y que ademas precisamos generar una gran cantidad de instancias, significaría estar desperdiciando espacio en memoria.

También para tener en cuenta: No existen reglas ni restricciones sobre los nombres de las funciones constructoras, pero existe una convención entre los programadores

de usar la letra capital en los nombres de las funciones constructoras (esto es, que la primera letra sea mayúscula).

7.1.3. Prototype class pattern

Para tratar de resolver el problema recién mencionado, en donde cada instancia tiene una copia del código de la función, es necesario hacer un buen uso del concepto de prototipo en JavaScript, el cual se hizo mención en la sección 2.6.4.

```
1  function Animal(nombre) {
2    this.nombre = nombre;
3  }
4
5  Animal.prototype.saludar = function() {
6    console.log('Hola, soy ', this.nombre);
7  };
8
9  var gato = new Animal('Garfield');
10 var perro = new Animal('Oddie');
11
12 perro.saludar();  // Hola, soy Garfield
13 gato.saludar();   // Hola, soy Oddie
```

Prototype class pattern

Para este caso, tendremos una función constructora `Animal`, la cual estará vinculada a su `[[Prototype]]`, un objeto que contendrá entre sus propiedades, a la función `saludar`. Bajo este patrón, nos obviamos de que cada instancia tenga una copia de la función saludar, y que gracias a la «cadena del prototipo» ese comportamiento esté delegado en `Animal.prototype`.

7.1.4. `class` en ES6

A partir de la versión ES6, una de las características más jugosas es la del uso de la palabra reservada `class`. Para desgracia del lector, ésta introducción al lenguaje no es más que *syntactic sugar*. Mediante una sintaxis más amena y amigable se alcanza la creación de clases, pero en el fondo la semántica es idéntica al Prototype class pattern.

```
1  class Animal {
2    constructor(nombre) {
3      this.nombre = nombre;
4    }
5    saludar() {
6      console.log('Hola, soy ', this.nombre);
7    }
8  }
9
10 var gato = new Animal('Garfield');
11 var perro = new Animal('Oddie');
12
13 perro.saludar();  // Hola, soy Garfield
14 gato.saludar();   // Hola, soy Oddie
```

Ejemplo de `class`

7.2. Herencia

Como se ha mencionado anteriormente, JavaScript tiene la particularidad de tener la herencia prototipada en vez de la herencia clásica. Este tipo de herencia es bastante poderosa, aunque también a veces incomprendida y mal aplicada.

7.2.1. Herencia simple mediante `prototype`

La herencia prototipal es tan potente que, haciendo un esfuerzo y algunos artilugios, se podrá simular la herencia clásica. La diferencia entre estos dos tipos de herencia es que en la herencia clásica, las subclases poseen una copia del comportamiento de clases. En la herencia prototipal no existe este concepto de copia. Un objeto tiene un prototipo y en caso de que el objeto no tenga un atributo o método necesario, delegará esta responsabilidad a su prototipo. Ésto se llama delegación de comportamiento.

Ahora, supongamos el siguiente código:

```
1  function Vehiculo(tipo) {
2    this.tipo = tipo;
3  }
4
5  Vehiculo.prototype.mostrarTipo = function() {
6    console.log(this.tipo);
7  }
8
9  function Auto(marca) {
10   Vehiculo.call(this, "terrestre");
11   this.marca = marca;
12 }
13
14 Auto.prototype = Object.create(Vehiculo.prototype);
15 Auto.prototype.constructor = Auto;
16
17 Auto.prototype.mostrarMarca = function () {
18   console.log(this.marca);
19 }
20
21 var fitito = new Auto("Fiat");
22 var falcon = new Auto("Ford");
23
24 fitito.mostrarMarca();      // Fiat
25 fitito.mostrarTipo();       // terrestre
26
27 falcon.mostrarMarca();      // Ford
28 falcon.mostrarTipo();       // terrestre
```

Analizando herencia prototipal en JS

Tal como se mencionó en la sección 7.1, las funciones en JavaScript son un mecanismo por naturaleza para simular las clases. ?t'Qué sucede en el código de arriba?

Se definen las funciones `Vehiculo` y `Auto` que serán nuestras clases. Ambas funciones actúan de constructoras. Al `[[Prototype]]` de `Vehiculo` se le agrega un método `mostrarTipo` mientras que al `[[Prototype]]` de `Auto` se le agrega `mostrarMarca`. Hasta ese punto es fácil de entender lo sucedido si se ha comprendido las secciones de prototype y de clases.

¿Qué pasa en las líneas 10, 14 y 15?

- Para el caso de la línea 10, la sentencia `Vehiculo.call(this, "terrestre")` está simulando una llamada al constructor padre. Lo que sería análogo a hacer `super` en Java. Ésta es una de las partes más feas del código, ya que esta llamada debe ser de forma manual, y dándo como parámetro el contexto del invocador.
- En la línea 14 estamos «pisando» el viejo objeto `Auto.prototype` que había sido creado al momento de declarar la función `Auto`, y creando un nuevo objeto. El método `Object.create` crea un nuevo objeto y establece como prototipo del objeto lo que haya sido pasado como primer argumento de la función.
- La línea 15 es la que quizás muchos programadores suelen omitir. Sin ella, si hacemos `console.log(fitito)` o `console.log(fitito.constructor)` podemos verificar que nos figura que `fitito` es `Vehiculo` en vez de `Auto`. ?t'Por qué sucede esto? En la línea 14, cuando hicimos la vinculación del prototype de `Auto` con el prototype de `Vehiculo`, creamos un nuevo objeto y hemos perdido la referencia de `Auto.prototype.constructor`.

En ejecución, podemos imaginarnos la siguiente imagen como una representación de lo que hay en memoria:

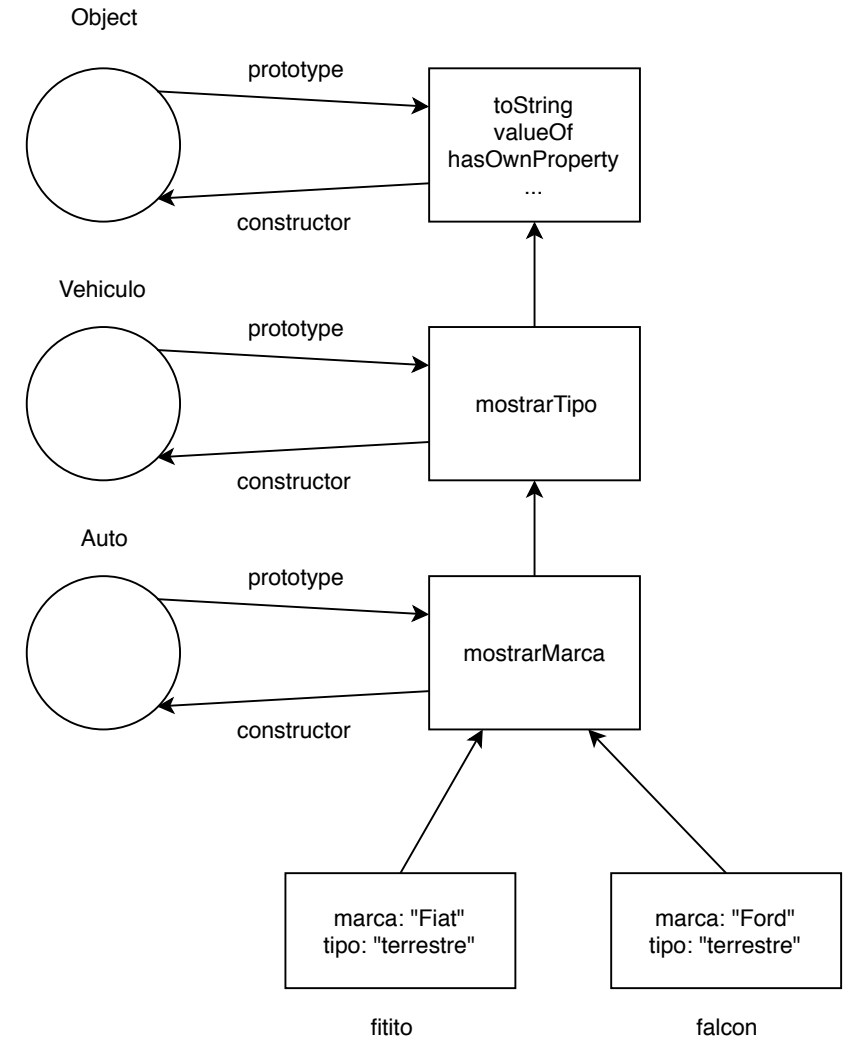

FIGURA 7.1: Diagrama del código presentado

7.2.2. `extends` en ES6

Tal como se explicó sobre la palabra reservada `class` en la sección 7.1.4, otra de las características que se introdujeron a partir de la versión ES6 es la de la palabra `extends` para simular la herencia de clases. Nuevamente ocurre lo mismo que lo mencionado anteriormente: Esta característica es meramente *syntactic sugar* que le omite al programador la necesidad de pensar en prototipos.

```
1  class Vehiculo {
2    constructor(tipo) {
3      this.tipo = tipo;
4    }
5    mostrarTipo() {
6      console.log(this.tipo);
7    }
8  }
9
10 class Auto extends Vehiculo {
```

```
11    constructor(marca) {
12      super("terrestre");
13      this.marca = marca;
14    }
15    mostrarMarca() {
16      console.log(this.marca);
17    }
18  }
19
20  var fitito = new Auto("Fiat");
21  var falcon = new Auto("Ford");
22
23  fitito.mostrarMarca();      // Fiat
24  fitito.mostrarTipo();       // terrestre
25
26  falcon.mostrarMarca();      // Ford
27  falcon.mostrarTipo();       // terrestre
```

`extends` en ES6

Para el programador que viene de C++ o Java, este uso de `class` y `extends` es, por lejos, muchísimo más amigable que crear funciones y vincularlas mediante sus prototipos. Otra de las introducciones a partir de ES6 es el uso del `super` en el constructor. Esto facilita enormemente la llamada al constructor de la superclase, o de métodos de la superclase, además de mimetizar la sintaxis de Java.

7.2.3. Herencia múltiple

Dado que los objetos «heredan» de un único prototipo, el lenguaje no provee ninguna herramienta natural para el soporte de la herencia múltiple. Otros lenguajes buscan alcanzar la herencia múltiple mediante el uso de interfaces. En JavaScript no existen las interfaces, pero sí existe una técnica llamada mixins (acrónimo para *mixed in*, del inglés «mezclado») para introducir un comportamiento a una clase sin necesidad de hacerla heredar de otra.

Una función de Mixin suele tener esta forma:

```
1  function mixin(fuente, destino) {
2    for (var prop in fuente) {
3      if (fuente.hasOwnProperty(prop)) {
4        destino[prop] = fuente[prop];
5      }
6    }
7  }
```

Función de Mixin

Lo ideal sería pensar en `fuente` como un objeto cuyas propiedades serán los miembros a «inyectar» en `destino`. Esta técnica llevó al estandar a agregar un método `Object.assign` a partir de ES6, el cual copia los valores de todas las propiedades enumerables en un objeto destino.

Otra manera de implementar los Mixins en ES6 es aprovechando que las clases son de primer órden. Éstas pueden ser pasadas como argumento o ser retornadas por una función.

```
var MyMixin = (superclass) => class extends superclass {
  saludar() {
    console.log('hola!');
  }
};

class Foo {}
class Bar extends MyMixin(Foo) {
  despedirse() {
    console.log('chau!');
  }
}

var objeto = new Bar();

objeto.saludar();      // hola!
objeto.despedirse();   // chau!
```

Haciendo uso de `class` como expresión

Como se puede observar, existen varias formas de aplicar esta técnica. Como se mencionó anteriormente, no hay soporte para herencia múltiple en JavaScript, aunque ésta es una forma de resolver el problema. Aún así, tiene sus defectos. El *shadowing* u *overriding* de métodos con el mismo identificador es una convención a tener en cuenta. También hay que pensar en el costo, ya que copiar las propiedades de un objeto a otro requiere de un esfuerzo.

7.3. Encapsulamiento

Dependiendo la perspectiva, se puede decir que el lenguaje tampoco provee un mecanismo natural para el encapsulamiento. Si hablamos de ocultar el estado de un objeto, es decir, de hacer sus datos miembro privados, no existe ninguna palabra reservada para ello. De hecho, todas las propiedades de un objeto son públicas.

Existe una convención extra oficial de que los miembros privados de un objeto comiencen su identificador con un guión bajo. Nuevamente, ésta no es una convención del lenguaje, sino de la comunidad. Más allá de la convención, volvemos a lo mismo: Por fuera se puede analizar cuál es el valor ligado a dicha propiedad.

```
class Persona {
  constructor(nombre, saldo) {
    this.nombre = nombre;
    this._saldo = saldo;
  }
}

var pepe = new Persona('Jose', 25);

console.log(pepe.nombre); // Jose
console.log(pepe._saldo); // 25
```

Descubriendo variables «privadas»

Como se puede observar en el ejemplo, se busca hacer privada la propiedad `_saldo`, pero sin embargo es visible desde afuera del objeto.

Por suerte mediante el uso de *closures*, se puede lograr el encapsulamiento, pero solo mediante el uso de funciones constructoras. Lamentablemente, para la sintaxis de `class` de ES6 no hay soporte nativo aún. Al momento de escribirse este documento, existe una propuesta en borrador para agregar al lenguaje, la cual se encuentra en *Stage 3* (para más información, ver).

```
1  function Persona(nombre, saldo) {
2    var saldoPrivado = saldo;
3    this.nombre = nombre;
4    this.obtenerSaldo = function() {
5      return saldoPrivado;
6    }
7    this.actualizarSaldo = function(nuevoSaldo) {
8      saldoPrivado = nuevoSaldo;
9    }
10 }
11
12 var pepe = new Persona('Jose', 25);
13
14 console.log(pepe.nombre);          // Jose
15 console.log(pepe.saldoPrivado);    // undefined
16 // OK, ya que es una propiedad privada.
17 console.log(pepe.obtenerSaldo()); // 25
18 pepe.actualizarSaldo(32);
19 console.log(pepe.obtenerSaldo()); // 32
```

Alcanzando variables privadas mediante closures

Un detalle menos obvio pero aún así importante, es que necesariamente los *setters* y *getters* de las variables ocultas por el closure deberán formar parte de la función constructora (es decir, no podrán definirse dentro del prototipo), lo que significa nuevamente que cada instancia de `Persona` tendrá código repetido, lo que implica gasto en memoria. Bajo esta situación, nos encontramos en un *trade-off* de tener datos miembro privados, pero no poder hacer uso correcto del patrón prototipal.

7.4. Polimorfismo

Quizás uno de los puntos más complicados de analizar en cuanto a JavaScript y su relación con el paradigma de orientación a objetos sea el de polimorfismo, dado que por su naturaleza de débilmente tipado y su particularidad de herencia prototipada, no resultará simple hacer una comparación con lenguajes como C++ o Java.

Para el caso de las funciones «polimórficas» el lenguaje no pone restricciones en relación a la aridad ni el tipo de los parámetros. Si la cantidad de argumentos dados al momento de una invocación es menor a la cantidad de parámetros formales de la función, entonces los restantes se considerarán con valor `undefined`.

Existe un identificador especial reservado para el vector de argumentos en JavaScript, bajo el identificador de `arguments`. Este es un objeto especial, aunque a simple vista parece un arreglo, no lo es. Funciona de una forma similar a lo que es `args` en

Java o `argv` en C++. Supongamos a continuación una función que no tiene definidos parámetros formales, pero aún así recibe argumentos a la hora de invocarla.

```
function mostrarArgumentos() {
  for (var i = 0; i < arguments.length; i++) {
    console.log(i + '. ' + arguments[i]);
  }
}

mostrarArgumentos('hola', 1, true, { a: 3 });

// 0. hola
// 1. 1
// 2. true
// 3. [object Object]
```

Analizando `arguments`

Para el caso de polimorfismo bajo una misma jerarquía de herencia, dado que el lenguaje es débilmente tipado, no posee las restricciones fuertes que posee un lenguaje como Java. Al ser interpretado, no existe un chequeo estático para corroborar que un método que esté siendo invocado pertenezca a un tipo o una clase particular. Ésta característica, la de redefinir un método de la superclase en una subclase, se llama polimorfismo de inclusión (o de subtipado).

```
class Animal {
  mover() {}
}
class Pez extends Animal {
  mover() {
    console.log("Soy pez y estoy nadando...");
  }
}
class Ave extends Animal {
  mover() {
    console.log("Soy ave y estoy volando...");
  }
}

var animales = [new Pez(), new Ave()];

for (let animal of animales) {
  animal.mover();
}

// Soy pez y estoy nadando...
// Soy ave y estoy volando...
```

¿Qué sucede exactamente?. En ejecución, cada elemento de la lista de `animales` hará búsqueda del método `mover` en su cadena de prototipo. En caso de no encontrarlo, resultará en un error en ejecución.

7.5. Modularidad

Durante sus primeros 20 años de vida, JavaScript no proveía soporte para módulos de una forma nativa. En 2015 con la salida de ES6, el lenguaje adquirió ese soporte nativo que le faltaba. Sin embargo, aún no todos los navegadores (o motores) soportan todas las funcionalidades introducidas en ES6 y las nuevas versiones del estándar.

En ésta sección vamos a mencionar cuáles son las formas en las que se alcanza la modularidad en JavaScript.

7.5.1. Módulos mediante patrones

Al igual que sucede con las clases, se hace el uso de IIFE y closures para aplicar patrones conocidos en la creación de módulos. El patrón por excelencia en este caso es el Revealing Module pattern. Si este patrón es implementado mediante IIFE, se lo puede pensar al módulo como un singleton, ya que al momento de definir el módulo se está creando una única instancia del mismo.

```
1  var ModuloSaludador = (function () {
2    var cantidadSaludos = 0;
3
4    var incrementarSaludos = function() {
5      cantidadSaludos++;
6    }
7
8    var saludar = function() {
9      console.log("hola");
10     incrementarSaludos();
11   }
12
13   var despedirse = function() {
14     console.log("chau");
15     incrementarSaludos();
16   }
17
18   var mostrarContador = function() {
19     console.log(cantidadSaludos);
20   }
21
22   return {
23     saludar: saludar,
24     despedirse: despedirse,
25     imprimirEstado: mostrarContador
26   }
27 }());
28
29 ModuloSaludador.saludar();         // hola
30 ModuloSaludador.despedirse();      // chau
31 ModuloSaludador.imprimirEstado(); // 2
```

Revealing module pattern

Para el ejemplo dado, se mantiene un estado interno que cuenta la cantidad de saludos dados. Se puede apreciar como tanto `cantidadSaludos` y la función `incrementarSaludos` son de alguna forma atributos privados del módulo.

Lo que retorna la función constructora del módulo en realidad es un objeto con las funciones del mismo, a modo de API. Notar el detalle de la función `mostrarContador`, que internamente para el módulo se llamará de esa manera, pero el módulo la expone con otro nombre, `imprimirEstado`.

La simplicidad de éste método de modularizar es una gran ventaja. No se requieren librerías externas. Una ventaja clara de la utilización de módulos es que podemos mantener `namespaces` más limpios a nivel aplicación.

Una desventaja de éste método es que no existe un manejo de dependencias entre módulos. Se puede hacer inyección de dependencias pasándole mediante parámetro el módulo que queremos inyectar como dependencia al nuevo módulo que estemos definiendo. Esto nos obliga a tener que pensar qué módulos deben estar definidos previamente a otros (recordar que el módulo es una IIFE), o nos obliga a cambiar un poco el patrón y utilizar algo más similar al Prototype class pattern, en donde podremos crear más de una instancia de un módulo en particular. Veremos en las soluciones siguientes cómo se aborda el manejo de dependencias para las otras técnicas.

7.5.2. Sistemas de módulos

Ante la falta de soporte, la propia comunidad se encargó de crear sus propios formatos estandarizados para modularizar sus aplicaciones. A mi entender, las dos partes claves que se agregaron con esta solución fue el manejo de dependencias entre módulos, y la capacidad de poder separar módulos en distintos archivos.

Los dos formatos más populares son **AMD** y **CommonJS**. Se pueden pensar a estos formatos como la parte sintáctica de los módulos, ya que luego será necesario hacer uso de algún module loader (librerías externas tales como **SystemJS** o **RequireJS**) para conectar y hacer funcionar a los módulos.

AMD

La sigla representa «Asynchronous Module Definition», en español sería definición de módulo asíncrono. Tal como se puede intuir por el nombre, se trata de una especificación para definir módulos y dependencias, y que las mismas sean cargadas de forma asincrónica.

En la especificación se puede encontrar un único método `define` para definir un módulo que, tiene sus variantes dependiendo de la cantidad y el tipo de los parámetros dados. Para los ejemplos que se mencionarán aquí, solo usaremos dos parámetros: el primero, un `Array` que representan las dependencias del módulo que estamos definiendo, y el segundo, una `Function` que será la definición del módulo propiamente dicho.

Un ejemplo de módulo con dependencias podría ser el siguiente:

```
1 define(['./math', './mailer'], function (math, mailer) {
2   var enviarSiEsPrimo = function(n, email) {
3     if (math.esPrimo(n)) {
4       mailer.enviar(email);
5     }
```

```
  }

  return {
    enviarSiEsPrimo: enviarSiEsPrimo
  }
})
```

Ejemplo de AMD

El primer parámetro dado es un arreglo con las dependencias. En nuestro caso, damos la ruta relativa a otros dos módulos *math* y *mailer* que supongamos que existen. El segundo parámetro es la definición del módulo que queremos crear, el cual será una función cuyos argumentos formales corresponden a las dos dependencias recién mencionadas. Lo que sucede por detrás es una inyección de éstas dependencias.

Para nuestro caso del ModuloContador, una adaptación en AMD podría ser la siguiente:

```
define([], function () {
  var cantidadSaludos = 0;

  var incrementarSaludos = function() {
    cantidadSaludos++;
  }

  var saludar = function() {
    console.log("hola");
    incrementarSaludos();
  }

  var despedirse = function() {
    console.log("chau");
    incrementarSaludos();
  }

  var mostrarContador = function() {
    console.log(cantidadSaludos);
  }

  return {
    saludar: saludar,
    despedirse: despedirse,
    imprimirEstado: mostrarContador
  };
});
```

Modulo contador en AMD

CommonJS

La otra especificación de módulos popular es la de CommonJS. A diferencia de AMD, la carga de los módulos se hace de forma sincrónica.

Podemos separar a la definición de un módulo en tres partes:

- La importación de las dependencias, que se hacen mediante un método especial `require`.

- La definición del módulo propiamente dicho, es decir, su código.
- La exportación de los métodos públicos del módulo, mediante uso de la propiedad `module.exports`.

Nuevamente, la idea no es profundizar sobre la especificación de CommonJS. De hecho, es un poco más amplia de lo que se menciona en éste documento, pero para el concepto que se quiere mostrar, es suficiente con lo recién mencionado.

Un ejemplo equivalente a lo realizado en el módulo AMD, el cual usaba dependencias *math* y *mailer*, podría ser el siguiente:

```
1  var math = require('./math');
2  var mailer = require('./mailer');
3
4  var enviarSiEsPrimo = function(n, email) {
5    if (math.esPrimo(n)) {
6      mailer.enviar(email);
7    }
8  };
9
10 module.exports.enviarSiEsPrimo = enviarSiEsPrimo;
```

Para el caso del ModuloContador, una versión en CommonJS podría ser la siguiente:

```
1  var cantidadSaludos = 0;
2
3  var incrementarSaludos = function() {
4    cantidadSaludos++;
5  };
6
7  var saludar = function() {
8    console.log('hola');
9    incrementarSaludos();
10 };
11
12 var despedirse = function() {
13   console.log('chau');
14   incrementarSaludos();
15 };
16
17 var mostrarContador = function() {
18   console.log(cantidadSaludos);
19 };
20
21 module.exports = {
22   saludar: saludar,
23   despedirse: despedirse,
24   imprimirEstado: mostrarContador
25 };
```

Modulo contador en CommonJS

7.5.3. Módulos en ES6

Probablemente una de las características más enriquecedoras introducidas a partir de ES6, es la del soporte nativo para los módulos. Este soporte de módulos mimetiza

la característica de asincronía en AMD, y la sintaxis concisa en CommonJS. De hecho, la sintaxis es extramadamente más concisa que en CommonJS, además de que se tiene un mejor soporte para las dependencias cíclicas, y gracias a la estructura de los módulos en ES6, se puede hacer un análisis estático y así realizar, por ejemplo, optimizaciones.

Las dos palabras resevadas a tener en cuenta para éste tipo de modularización son `import` y `export`. Con `import` se realizará la importación de dependencias para el módulo que estemos definiendo. Con `export`, se realizará la exportación cualquier miembro del módulo que se desee exponer. Una vez más, existen variantes para la parte de `export` como por ejemplo *named export* y *default export* que si bien se las mencionarán mediante ejemplos, escapan de este documento y queda a cargo del lector conocerlas en detalle.

Un dato a tener en cuenta, es que el soporte de los módulos para ES6 aún no está implementado en su completitud en el intérprete de todos los navegadores. Al punto tal de que quizás para hacer uso de ésta característica puede que sea necesaria una herramienta de traducción y compilación (acción conocida como «transpile») tal como **Babel**, y una herramienta de compilación o empaquetamiento tal como **Webpack**.

Tomando como caso de ejemplo el supuesto módulo *math* visto en las secciones anteriores, podríamos hacer `import` de éste módulo dependiendo de la forma en la que se esté exportando.

```
1 // importando lo que esté por "default"
2 import math from './math';
3 // importando todo lo exportado por math
4 import * as math from './math';
5 // importando y haciendo destructuring de lo que nos sirva
6 import { esPrimo } from './math'
```

Algunos ejemplos de `import`

Para el caso de la exportación también existen varias maneras, y dependiendo de cuál se utilice, será correspondiente luego la forma en la que se importe.

```
1 // exportando una constante con nombre
2 export const PI = 3.1416;
3 // exportando una funcion
4 export function foo() { console.log('foo') }
5 // exportando con "default"
6 export default function bar() {}
```

Algunos ejemplos de `export`

Dependiendo cual sea el caso, tiene sentido después pensar qué parte del módulo se está importando: Si tan solo una parte, el módulo completo, alguna función o atributo particular (siempre que se esté exportando), o lo que exporta el módulo por defecto.

Siguiendo con el grupo de ejemplos, se muestra cómo sería una implementación en ES6 de los módulos ejemplificados en las secciones anteriores:

```
1 import { esPrimo } from './math';
2 import { enviar } from './mailer';
3
4 export function enviarSiEsPrimo(n, email) {
5   if (esPrimo(n)) {
6     enviar(email);
7   }
8 }
```

Ejemplo de módulo en ES6

Un particularidad a tener en cuenta en este ejemplo es que en las lineas 1 y 2 se hizo *destructuring* de los módulos *math* y *mailer*, dado que solo nos interesan las funciones `esPrimo` y `enviar`. Por otro lado, el `export` en la definición de la función se puede hacer de forma separada (es decir, definir la función por un lado y luego hacer `export enviarSiEsPrimo`. Cualquiera de las dos formas es válida, pero se busca mostrar qué tan conciso queda el código con ES6.

Ahora veamos un ejemplo de ModuloContador, el cual no posee dependencias. Para este caso, mostraremos una variante del `export` totalmente válida, en la que se hace un único `export` de las tres funciones que expone el módulo.

```
1  var cantidadSaludos = 0;
2
3  var incrementarSaludos = function() {
4    cantidadSaludos++;
5  };
6
7  var saludar = function() {
8    console.log('hola');
9    incrementarSaludos();
10 };
11
12 var despedirse = function() {
13   console.log('chau');
14   incrementarSaludos();
15 };
16
17 var mostrarContador = function() {
18   console.log(cantidadSaludos);
19 };
20
21 export { saludar, despedirse, mostrarContador as imprimirEstado };
```

Modulo contador en ES6

7.6. Conclusiones

- Lo bueno: La herencia prototipal y su poder. El modelo de delegación de comportamiento y la cadena de prototipo, y la falta de necesidad de copiar o guardar lugar en memoria para los miembros de la superclase seguramente lo hacen más liviano. La sintaxis de clase de ES6 fue probablemente una de las mejores características lanzadas para el lenguaje. Permite a muchos programadores meterse en el lenguaje sin necesidad de cambiarles la mentalidad de herencia

clásica a la prototipal. Dicho todo esto, JavaScript es un verdadero lenguaje orientado a objetos, mientras que otros lenguajes son orientados a clases. El polimorfismo (o en realidad la falta de chequeos en «compilación») es otro punto a destacar. Le saca rigurosidad (y seguridad) al lenguaje incrementando su flexibilidad. Si quiero invocar a un método de un objeto y éste no existe, se buscará en la cadena de prototipo hasta terminar con la cadena, y en caso de que así sea, habrá un error en ejecución. Otra característica que también merece una mención es la de los módulos de ES6, ya que de una manera clara y concisa se pueden separar espacios de nombres y manejar las dependencias sin la necesidad de pensar en ellas.

- Lo malo: La falta de soporte natural para las clases es algo que deja que desear del lenguaje. Para ser justos con él, no fue pensado para tener clases (y es por eso que no tiene herencia clásica), y aún así los programadores buscan llegar a las mismas. Sin embargo, el soporte dado a las clases en ES6 es únicamente sintáctico, y el estándar parece estar conforme con dichas bases como para seguir evolucionando en éste punto. La asignación «manual» del prototipo de una función al querer simular la herencia clásica es un arma de doble filo, ya que si bien nos da libertades, es muy fácil perderse entre las relaciones que hay entre los objetos. Por otro lado, la poca popularidad de la herencia prototipal hace de JavaScript un lenguaje más difícil de comprender.
- Lo feo: No poder poseer miembros privados, ya sean atributos o funciones, y tener que recurrir a aplicar mecanismos como closures e IIFEs para alcanzar esto. Todos los métodos y los atributos cargados en un objeto serán públicos y habrá que pensar en un buen diseño de aplicación para no tener problema con ello. Por suerte, los módulos de ES6 solucionan una parte de esto, ya que de forma implícita, los métodos que no se exporten en un módulo valdrán como métodos privados para ese módulo.

Bibliografía

Crockford, Douglas (2008). *JavaScript: The Good Parts*. O'Reilly Media, Inc. ISBN: 0596517742.

Rauschmayer, Axel (2014). *Speaking JavaScript*. 1st. O'Reilly Media, Inc. ISBN: 1449365035, 9781449365035.

Resig, John, Bear Bibeault y Josip Maras (2016). *Secrets of the JavaScript Ninja*. 2nd. Greenwich, CT, USA: Manning Publications Co. ISBN: 1617292850, 9781617292859.

Simpson, Kyle (2014a). *You Don'T Know JS: Scope & Closures*. 1st. O'Reilly Media, Inc. ISBN: 1449335586, 9781449335588.

– (2014b). *You Don'T Know JS: This & Object Prototypes*. 1st. O'Reilly Media, Inc. ISBN: 1491904151, 9781491904152.

– (2014c). *You Don'T Know JS: Types & Grammar*. 1st. O'Reilly Media, Inc. ISBN: 1491904194, 9781491904190.