

Clase N°1 - Training Node.js

¿Qué es JavaScript?

En simples palabras, es un lenguaje de scripting de propósito general, que se ajusta a la especificación [ECMAScript](#). Este fue originalmente diseñado para programas de poca escala dentro del navegador web, pero el JavaScript moderno que conocemos hoy en día es radicalmente diferente. Originariamente, este fue creado por Netscape. De hecho, el nombre “JavaScript” es una marca registrada de Sun Microsystems (Hoy en día conocido como Oracle).

Explorando JavaScript

La mejor forma de aprender un nuevo lenguaje o adquirir un mejor manejo de este, es mediante la práctica. Recomendamos realizar los ejercicios prácticos a medida que avanzamos con los temas teóricos. Lo único que necesitaremos será un interpretador para poder efectuar las pruebas correspondientes, de lo contrario podremos proceder directamente con ejecutar el código con Node.js

Clases

JavaScript es un lenguaje basado en prototipos y cada objeto en él tiene una propiedad interna oculta llamada “prototype” que puede usarse para extender propiedades y métodos de objetos.

Con la llegada de la especificación de lenguaje ES6 se introduce una sintaxis que nos permitirá poder definir clases más elegante y fácilmente.

```
class AnyName {
  property1;
  property2;

  constructor(...args) {}

  get foo() {
    return this.foo;
  }

  set foo(value) {
    this.foo = value;
  }

  static staticMethod() {
    return 'I am a static method';
  }

  publicMethod() {
    return 'I am a public method';
  }
}
```

Pero recordemos, en Javascript no existe el concepto de *clases*. JavaScript es un lenguaje prototipado. Aunque uses la nueva sintaxis para crear "*clases*", estarás creando funciones con un prototipo.

Dentro de una clase podemos encontrar propiedades, su constructor, getters y setters, así como métodos estáticos y públicos.

Constructor

El constructor es el método encargado de crear nuestro objeto, e inicializar sus propiedades, cuando instanciamos una clase, automáticamente se ejecuta el constructor.

Métodos públicos

Podemos generar los métodos que sean necesarios para nuestra clase, y van a ser los métodos disponibles para invocarlos en nuestra aplicación.

Métodos privados

Con JavaScript existen varias formas de emular métodos privados:

- Mediante el uso de funciones globales
- Encapsulando la lógica dentro del constructor
- Mediante el operador de enlace

Aunque en una próxima especificación se permitirá **definir miembros privados en las clases**, de momento no disponemos de un mecanismo para ocultar contenido.

Tipos primitivos y compuestos

El último estándar ECMAScript define nueve tipos:

Tipos de datos primitivos

- Undefined
- Boolean
- Number
- String
- BigInt
- Symbol

Tipos de datos compuestos

El tipo compuesto es un cuerpo de datos compuesto por varios tipos de datos básicos (y también puede incluir tipos compuestos). Existen aproximadamente tres tipos de tipos compuestos en JavaScript de la siguiente manera:

- Objeto
- Matriz
- Función

Constantes y variables

A continuación, aprenderemos algunos de los conceptos más importantes que nos ofrece este lenguaje para la declaración de variables y constantes. Para ello, recurriremos a algunos casos prácticos y errores frecuentes al momento de codificar.

Scopes y closures

Cada variable que declaramos tiene un cierto nivel de visibilidad que determina dónde podemos utilizarla. Esto significa que al declarar una variable, no significa que esta pueda ser accedida desde cualquier parte de nuestro código.

En términos formales, el scope puede definirse como el alcance que una variable tendrá en el código. En otras palabras, el scope decide a qué variables tienes acceso en cada parte del código.

Scope global

```
let counter = 0;

function returnCount() {
  return counter;
}

returnCount();
```

En este ejemplo podemos ver como la variable counter es declarada fuera de la función returnCount(). Esta distinción, significa que la variable counter puede ser accedida en cualquier parte del código.

Scope Local

Cada variable declarada dentro de una función no puede ser accedida fuera de esta, tal cual como muestra la imagen a continuación:

```
let state;

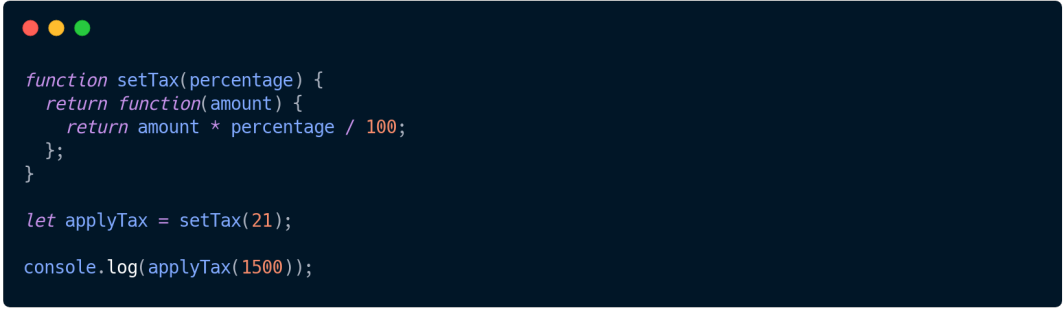
function setState() {
  let state = "on";
}

setState();

console.log(state); // Undefined
```

Closures

Cuando nos acercamos por primera vez al concepto de closure es habitual encontrarnos con problemas a la hora de entenderlo, a pesar de que seguramente en muchas ocasiones los habremos utilizado sin saberlo.



```
function setTax(percentage) {  
  return function(amount) {  
    return amount * percentage / 100;  
  };  
}  
  
let applyTax = setTax(21);  
  
console.log(applyTax(1500));
```

A pesar de que las variables que esta función utiliza se encuentran en otro ámbito en el momento de su ejecución, JavaScript guardó una referencia al valor de las mismas y por lo tanto siempre están accesibles para la función.

Arreglos

Un arreglo es una colección de valores, en donde cada uno de estos es denominado elemento, y cada elemento posee una posición numérica conocida como índice. Cada elemento de un arreglo puede ser de un tipo diferente. Estos pueden ser a su vez objetos u otro arreglo, permitiéndonos crear estructuras de datos complejos. El tamaño de estos es dinámico y no es necesario especificarlo al crearlo o a medida que va mutando.

En los ejemplos prácticos veremos las diferentes formas de crear un arreglo.

Objetos

Un objeto en JavaScript es una colección de propiedades, y una propiedad es una asociación entre un nombre o clave y un valor. El valor de una propiedad puede ser una función o método.

Para comprender este concepto, nuevamente recurriremos a algunos ejemplos prácticos.

Operador de propagación

También conocido en inglés como spread operator, permite que una expresión sea expandida en situaciones donde se esperan múltiples argumentos o múltiples elementos.

Estructuras condicionales

Cada decisión que tomemos puede ser contestada mediante un operador lógico. Dicha respuesta, determinará el siguiente paso que efectuaremos.

Can be any expression that evaluates to a `true` or `false`

```
if (something_is_true) {  
    do_something;  
} else {  
    do_something_different;  
}
```

En las mayorías de los casos, nuestra expresión será un simple valor lógico. En el caso que deseemos de realizar una evaluación de una variable, contamos con algunos operadores.

```
if (expression operator expression) {  
    do_something;  
} else {  
    do_something_different;  
}
```

Otras expresiones que podemos utilizar para la evaluación de múltiples elementos, puede ser mediante la **concatenación de if** o el uso del **switch**.

Ciclos de iteración

Gran parte de las veces, cuando codificamos necesitamos ejecutar cierto código de forma reiterativa. Para esto, en JavaScript contamos con diferentes sentencias:

- * Bucles mediante la expresión `for`
- * Bucles mediante la expresión `while`
- * Bucles mediante la expresión `do...while`

```
for (start_point; condition; step) {  
    // Code to execute  
}  
  
while (condition) {  
    // Code to execute  
}  
  
do {  
    // Code to execute  
} while (condition);
```

Funciones flecha

A partir de la especificación de ES6, se introduce un nuevo sintaxis para la escritura de las funciones. Este nuevo método posee una sintaxis más acotada y amigable a la hora de codificar. No posee grandes diferencias de la forma tradicional de escribir las funciones. A continuación, brindamos un ejemplo para una mejor comprensión.

```
// Common way to write functions
const circleArea = function(pi, r) {
  return pi * r * r;
}

const result = circleArea(3.14, 3);

// New syntax
const circleArea = (pi, r) => pi * r * r;
const result = circleArea(3.14, 3);
```

TypeScript

Como vimos, JavaScript no es un lenguaje difícil de aprender y nos ofrece cierta flexibilidad, pero este nos presenta ciertos desafíos cuando nuestras implementaciones comienzan a complejizarse y crecer a gran escala. Uno de estos desafíos, es que JavaScript es un lenguaje interpretado y por lo tanto no posee un paso de compilación previo. En el único momento que nosotros sabemos que existe un simple error de sintaxis en el código es en tiempo de ejecución. Otro desafío es que este no es un lenguaje orientado a objetos.

Mediante el uso de TypeScript podremos generar código JavaScript fuertemente tipado y orientado a objetos mediante el uso de un compilador.

¿Porqué programar en TypeScript?

Mediante el uso de este, podremos encontrar errores tempranamente en el código. Otro de los beneficios es que permite generar código mucho más limpio, escalable y consistente.

```
1 function addNumbers(x, y) {
2   return x + y;
3 }
4 console.log(addNumbers(3, "0"));
```

(a) The buggy program.

```
1 function addNumbers(x, y) {
2   return x + y;
3 }
4 console.log(addNumbers(3, 0));
```

(b) The fixed program.

```
1 function addNumbers(x:number, y:number) {
2   return x + y;
3 }
4 console.log(addNumbers(3, "0"));
```

(c) The annotated, buggy program.

Fig. 2: JavaScript coerces 3 to "3" and prints "30". From the fix, we learn that this behavior was unintended and add annotations that allow Flow and TypeScript to detect it.

¿Cómo compilar el código TypeScript en JavaScript?

El procedimiento es muy sencillo. Inicialmente, deberemos de contar con TypeScript instalado localmente para poder compilar el código posteriormente. Para esto, recomiendo revisar la [documentación oficial](#).

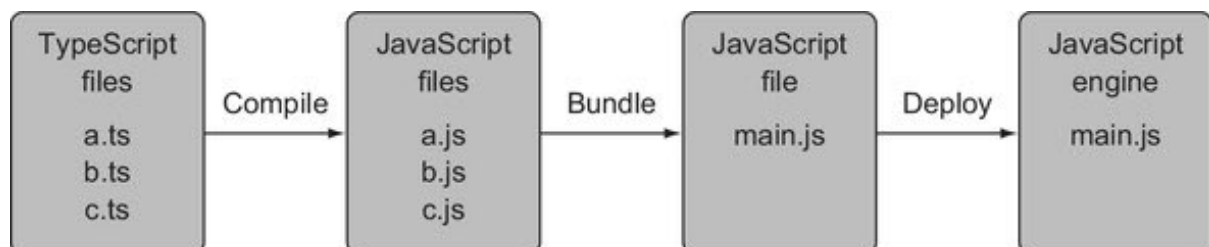
Instalando TypeScript

Mediante npm ejecutamos el siguiente comando “npm install -g typescript”. Una vez instalado, podemos lanzar el comando “tsc -v” para corroborar la versión instalada.

En el caso que deseemos adicionar una configuración al momento de compilar, podemos adicionar el archivo “tsconfig.json” con ciertos atributos como se muestra a continuación:

```
{
  "compilerOptions": {
    "baseUrl": "src",
    "outDir": "./dist",
    "noEmitOnError": true,
    "target": "es5"
  }
}
```

Flujo de trabajo con TypeScript



El bundle del código JavaScript puede ser efectuado mediante Webpack o Rollup que no simplemente concatena múltiple archivos en uno solo, sino que además puede optimizar el código y remover código sin uso.

Enlaces de interés

- [To type or not to type: quantifying detectable bugs in JavaScript](#)
- [Aprende TypeScript de 0 a 100 \(Gratis\)](#)

Libros recomendados para la lectura

- [TypeScript Quickly](#)