Bloque 5

Programación con JavaScript

Geolocation. Local Storage. TypeScript.

Programación con JavaScript
Cefire 2017/2018
Autor: Arturo Bernal Mayordomo

Índice

API de almacenamiento local (Local Storage)

Para almacenar datos de manera persistente en el cliente, tradicionalmente se han utilizado cookies. Sin embargo, tienen bastantes limitaciones, siendo una de ellas que no se envían cuando el servidor está ejecutándose en un dominio (o puerto) diferente al de la aplicación cliente.

Hoy en día, todos los navegadores soportan formas más avanzadas de almacenamiento, y el estándar básico hoy en día es la API de Local Storage. Existen otros sistemas más avanzados como WebSQL (soportada por Chrome, Opera o Safari, pero abandonada), o IndexedDB (estándar W3C y soportado por casi todos los navegadores).

Este sistema de almacenamiento es muy sencillo de usar. Si el navegador lo soporta, existirá un objeto global (dentro de window) llamado **localStorage**. A partir de dicho objeto podremos almacenar datos que estarán relacionados con el servidor donde se está ejecutando la aplicación cliente (se pueden repetir nombres de variables en diferentes dominios, no se solapan).

La funcionalidad de este objeto es muy sencilla:

```
// Creamos la variable usuario con el valor "Pepe"
localStorage.setItem("usuario", "Pepe");
// También se puede crear así:
localStorage.usuario = "Pepe";

//Obtener el valor guardado
console.log(localStorage.getItem("usuario")); // Imprime "Pepe"
// O así
console.log(localStorage.usuario);

// Borrar una variable/entrada
localStorage.removeItem("usuario");

// Borramos todos los datos almacenados
localStorage.clear();
```

Cabe recordar que existe persistencia de datos. A no ser que borremos los datos del navegador, la próxima vez que accedamos a nuestra página (y en cada recarga), los datos guardados seguirán ahí. Si queremos datos que se borren al cerrar la página (sólo para la sesión actual), podemos usar **sessionStorage** (se usa de forma idéntica).

API de geolocalización (Geolocation)

El API de Geolocalización presente en la mayoría de navegadores actuales (escritorio y móvil) permite usar JavaScript para consultar las coordenadas del geolocalización del usuario (mucho más preciso cuando está presente un sistema GPS → teléfono móvil). Es una funcionalidad muy importante en la actualidad, ya que te permite posicionarte en el mapa, encontrar usuarios que estén cerca, medir distancias, etc. Lo ideal es combinarla con una API de mapas como Google Maps.

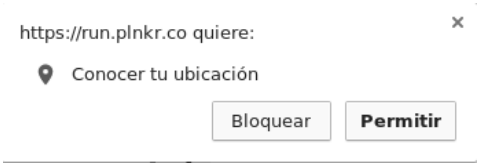
La localización consiste en dos componentes de coordenadas: **latitud**, que es la distancia hacia el norte (positiva) o sur (negativa). Y **longitud**, que es la distancia hacia el este (positiva) o al oeste (negativa) a partir del Meridiano de Greenwich.

Usamos **navigator.geolocation.getCurrentPosition** para geolocalizarnos. Este método es asíncrono (se ejecuta en segundo plano), por tanto no nos devuelve nada de forma inmediata. Le pasaremos una función que será ejecutada cuando el navegador encuentre finalmente la localización. Esta función recibirá un parámetro que contendrá la información de dicha localización.

Archivo: geolocation1.html

```
<!DOCTYPE html>
<html>
  <head>
     <title>Ejemplo de cómo obtener la geolocalización</title>
     <link rel="stylesheet" href="style.css">
  </head>
  <body>
     <div>
        <p id="coordenadas">Obteniendo información de la localización...</p>
     </div>
    <script src="geolocation1.js"></script>
  </body>
</html>
```

Archivo: geolocation1.js




```
navigator.geolocation.getCurrentPosition(function(pos) {
    var p = document.getElementById("coordenadas");
    p.textContent = "Latitud: " + pos.coords.latitude + ". Longitud: " + pos.coords.longitude  +
                    " (Precisión: " + pos.coords.accuracy + ")";
});
```

No olvides darle permisos al navegador para obtener tu localización.

Para gestionar errores, podemos pasarle una segunda función al método de geolocalización. Esta función será llamada en lugar de la primera si se ha producido un error, y recibirá por parámetro un objeto con una propiedad llamada **code** que contiene el tipo de error producido.

```
let p = document.getElementById("coordenadas");

navigator.geolocation.getCurrentPosition(function(pos) { // Función cuando todo va bien
  p.textContent = "Latitud: " + pos.coords.latitude + ". Longitud: " + pos.coords.longitude
```

```
            + " (Precisión: " + pos.coords.accuracy + ")";
}, function(error) { // Función cuando hay un error
   switch(error.code) {
      case error.PERMISSION_DENIED: // El usuario no permite que el navegador acceda a la localización
         p.textContent = "El usuario ha denegado la petición de geolocalización"
         break;
      case error.POSITION_UNAVAILABLE: // No puede obtener la localización
         p.textContent = "La información de localización no está disponible."
         break;
      case error.TIMEOUT: // Ha expirado el tiempo máximo para obtener la localización
         p.textContent = "Ha expirado el tiempo máximo para obtener la localización"
         break;
      case error.UNKNOWN_ERROR:
         p.textContent = "Se ha producido un error desconocido."
         break;
   }
});
```

Estas son las propiedades de localización que podemos obtener (algunas de estas sólo estarán disponible cuando usemos un GPS, como por ejemplo, en un móvil.):

- **coords.latitude** → Latitud, número decimal.
- **coords.longitude** → Longitud, un número decimal.
- **coords.accuracy** → La precisión, en metros.
- **coords.altitude** → Altitud, en metros sobre el nivel del mar (si está disponible).
- **coords.altitudeAccuracy** → La precisión de la altitud (si está disponible).
- **coords.heading** → La orientación en grados(si está disponible).
- **coords.speed** → La velocidad en metros/segundos (si está disponible)
- **timestamp** → El tiempo de respuesta, UNIX timestamp (si está disponible).

Opciones de la posición

Hay un tercer parámetro que podemos pasarle al método **getCurrentPosition**. Este parámetro es un objeto JSON que contiene una o más de estas propiedades:

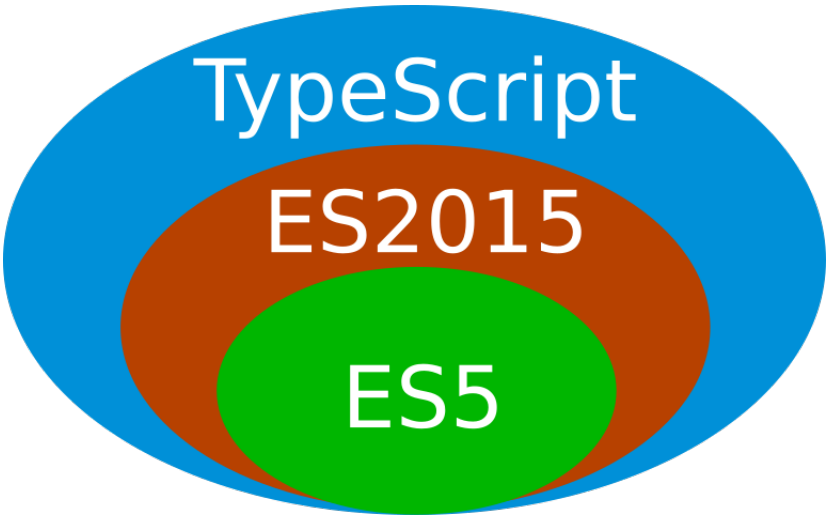
- **enableHighAccuracy** → Un booleano que indica si el dispositivo debería usar todo lo posible para obtener la posición con mayor precisión (por defecto false). Esta opción consume más batería y tiempo.
- **timeout** → Tiempo en milisegundos a esperar para obtener la posición o lanzará un error. Por defecto es 0 (espera indefinidamente).
- **maximumAge** → Tiempo en milisegundos que guarda la última posición el navegador en caché. Si se solicita una nueva posición antes de expirar el tiempo, el navegador devuelve directamente el dato almacenado en caché.

```
navigator.geolocation.getCurrentPosition(
   positionFunction,
   errorFunction,
   {
      enableHighAccuracy: true,
      timeout: 6000, // 6 segundos de plazo máximo
      maximumAge: 30000 // Puede ser una posición guardada en caché hace menos de 30 segundos
   }
);
```

Introducción a TypeScript

Para un gran número de aplicaciones, JavaScript quizás no sea el lenguaje más apropiado. Puede serlo si somos muy cuidadosos y separamos nuestra aplicación en clases y módulos utilizando las últimas características de ES2015, pero muchos desarrolladores tienden a escribir código espagueti (largos archivos con mucha funcionalidad mezclada), y no tienen cuidado con los tipos de datos que se le envían a las funciones (usando == en lugar de ===, por ejemplo, simplemente por no preocuparse por los tipos de datos, lo cual es un error).

TypeScript es un superconjunto de JavaScript. Eso significa que tiene todas las características que tiene JavaScript (la última versión incluida), además del tipado de variables, y otras que formarán parte de JavaScript en el futuro.



TypeScript es un lenguaje muy accesible para desarrolladores que ya están familiarizados con lenguajes como Java o C#. Además, no hay problemas de compatibilidad con navegadores (o entornos servidor como NodeJS), ya que se compila (transpila) a JavaScript puro (lo haremos con Webpack).

Otros lenguajes que compilan a JavaScript son CoffeeScript y Dart, pero TypeScript es el más próximo a JavaScript (como ya hemos mencionado, es un superconjunto). De hecho, si copiamos y pegamos JavaScript dentro de un archivo TypeScript (.ts), funcionará perfectamente.

Conociendo la versión ES2015 y posteriores, aprender TypeScript es muy fácil, ya que hay pocas diferencias entre ambos. De hecho, las diferencias más notables son que TypeScript soporta tipos (declarar una variable o una función de tipo número, string, booleano, objeto, etc.) y muchas características avanzadas de los Lenguajes de Programación Orientados a Objetos, como el ámbito público y privado de métodos y propiedades, propiedades opcionales, interfaces, etc.

Compilando TypeScript con Webpack

Si por curiosidad, queremos ver el código JavaSscript que se generaría cuando programamos en TypeScript, podemos probar escribiendo código y viendo el resultado en https://www.typescriptlang.org/play/index.html. No os preocupéis por el código generado, es el navegador el que lo tiene que interpretar, no vosotros.

Integrar TypeScript con Webpack (igual que hicimos con Babel), es bastante sencillo. Para ello debemos asegurarnos que tenemos instalado el compilador de TypeScript y un *loader* para Webpack.

```
npm install -D typescript

npm install -D ts-loader
```

Posteriormente, debemos configurar el archivo webpack.config.js tal como indica la página del loader, adaptándolo a nuestras necesidades:

```
module.exports = {
  devtool: 'inline-source-map' // Genera un sourcemap para depurar TypeScript desde el navegador
  context: path.resolve(__dirname, 'src'), // Nuestros archivos TypeScript están en el directorio src
  entry: {
    index: './index.ts' // Archivo principal (punto de entrada)
  },
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'), // Archivos compilados en src/ (lo que importamos en el HTML).
  },
  resolve: {
    extensions: ['.ts', '.tsx', '.js'] // Así no necesitamos poner la extensión al importar módulos
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        exclude: /node_modules/,
        loader: 'ts-loader',
      },
    ]
  },
  ...
};
```

Para sobrescribir las opciones por defecto del compilador de TypeScript, podemos crear un archivo llamado **tsconfig.json** en la ráiz del proyecto. Algunos ejemplos aquí y opciones disponibles aquí. Un ejemplo:

```
{
  "compilerOptions": {
    "target": "es5", // Compilamos a ES5
    "moduleResolution": "node", // Los import/export se traducen a formato NodeJS (require)
    "sourceMap": true, // Producir sourcemaps para depurar archivos TypeScript
    "removeComments": false, // Mantener comentarios (para depuración, el producción se eliminan)
    "alwaysStrict": true // Genera la instrucción “use strict” en los archivos JS producidos.
  }
}
```

Valores tipados

En TypeScript, todas las variables están tipadas por defecto (number, boolean, string, *nombre de clase*, …), por tanto, el compilador no nos permitirá asignarle posteriormente un valor de tipo diferente. Podemos usar el tipo especial **any**, que nos permite que una variable se comporte como en JavaScript (sin tipo definido).

```
let a: any; // La variable a puede tener cualquier valor de cualquier tipo
let b; // Igual que  a: cualquier tipo (por omisión es any).
a= 3;
a = "asdf";
b = 5;
b = true;
```

Si queremos forzar una variable para que contenga valores de un tipo específico podemos hacerlo de dos formas: declarar la variable con **nombre:tipo**, o asignar directamente un valor cuando la declaramos.

```
let a: number; // La variable a puede solo contener números
let b = 4; // Igual que b: números (tipo implícito en la asignación).
a= 3; // OK
a = "asdf"; // Error: 'string' not assignable to 'number'
b = true; // Error
let c: number = 12; // OK (tipo y asignación al mismo tiempo, aunque es redundante)
let d: number = "hello"; // Error
```

Podemos declarar los tipos de datos que se le pasan a una función o método y lo que nos devuelve. El compilador nos dará un error si no se cumple lo prometido.

```
class Person {
   private name: string;

   constructor(name: string) { // Recibe un parámetro de tipo string
      this.name = name;
   }

   getName() : string { // Devuelve un valor string
      return this.name;
   }
}
```

Tipo Array

Como los arrays son objetos de la clase **Array**. Es válido declararlos como **tipo[]**, o como **Array<tipo>** (siendo ***tipo*** el tipo de dato que contiene el array). En este caso estamos limitando a que en el array sólo puede haber elementos de un tipo de datos.

```
let a: Array<string>;
let b: string[];
a = ["hello", "goodbye"];
b = ["a", "b", "c"];
```

Tipo Object

Aunque es mejor usar interfaces, como pronto veremos, para definir las propiedades de un objeto, podemos tipar una variable indicando las propiedades que tendrá el objeto asignado (y su tipo):

```
// Recibe un parámetro que es un objeto con las propiedades length (number) y str (string)
function isRightLength (obj: { length: number, str: string}): boolean {
   return obj.str.length == obj.length;
}

let o : { length: number, str: string} = { // La variable contendrá un objeto con estas propiedades
   length: 5,
   str: "hello"
};
console.log(isRightLength(o)); // true
```

Las funciones lambda funcionan exactamente igual. A veces no es necesario definir el tipo de datos que devuelve una función. El compilador en este caso, analizará la sentencia return para saber qué devuelve.

```
// (obj: { length: number, str: string}) : boolean no es necesario (devuelve una condición → boolean).
let isRightLengthArrow = (obj: { length: number, str: string}) => obj.str.length == obj.length;;
console.log(isRightLengthArrow(o)); // true
```

Podemos también declarar un parámetro como opcional (puede no estar en el objeto) mediante el símbolo ? Tras el nombre del parámetro.

```
function areaRect(rect: {a: number, b?: number}): number { // La propiedad “b” es opcional
   if(!rect.b) return rect.a * rect.a;
   return rect.a * rect.b;
}

console.log(areaRect({ a: 10 })); // Imprime 100. El objeto no contiene la propiedad “b” pero es opcional
console.log(areaRect({ a: 5, b: 8 })); // Imprime 40
```

Tipo Function

Podemos asignar como tipo de una variable una función. Esto se representa con un tipo de construcción parecida a una lambda. Entre paréntesis irán los parámetros que se reciben y después de la flecha el tipo de valor que devolverá. Esa variable sólo podrá referenciar una función con las mismas características. Usa **void** como tipo de dato cuando no devuelva nada.

```
let func : (str: string, num: number) => string; // Dos parámetros (string, number) y devuelve un string
func = function (str, num) {
   let result = ""; // No es necesario decirle al compilador que es un string (implícito)
   for(let i = 0; i < num; i++) result += str;
   return result; // Debe devolver un string, en el caso contrario no compilará
}
console.log( func("a", 6) ); // Imprime "aaaaaa" (debe ser llamado con un string y un number o no compilará)
```

Clases e interfaces en TypeScript

Las clases en TypeScript son muy parecidas a las clases en ES2015, con más características. Una de las principales diferencias es que podemos establecer las propiedades y métodos como **públicos** o **privados** (en ES2015 todo es público). Además, declararemos las propiedades fuera del constructor:

```
class Rect {
   private width: number = 5;
   private height: number = 8;

   public getArea(): number { // 'public' puede ser omitida (por defecto todo es public)
      return this.width * this.height;
   }
}

let rect = new Rect();
rect.width; // Error: Property 'width' is private and only accessible within class 'Rect'
```

Constructor

El método del constructor es igual al que usamos en ES2015, pero tiene otra característica. Podemos crear propiedades directamente en los parámetros del constructor. Para ello, ponemos la palabra reservada **public** o **private** delante del parámetro. Internamente se creará una propiedad con el mismo nombre en el objeto y se le asignará el valor (**this.propiedad = propiedad**).

```
class Rect {
   constructor(public width: number, private height: number) { } // propiedades creadas y asignadas

   public getArea(): number {
      return this.width * this.height;
   }
}

let rect = new Rect(5, 7);
console.log(rect.getArea()); // Imprime 35
```

Getters y setters

Hay dos modos para crear getters y setters. La forma clásica es tener un atributo privado, y dos métodos llamados getAttribute y setAttribute para obtener y modificar el valor de forma controlada:

```
class Person {
   constructor(private name: string) {}

   getName() : string { // Devuelve un string
      return this.name;
   }

   setName(name: string) { // Devuelve un string
      this.name = name;
   }
}
```

Y la equivalente a usar **Object.defineProperty** en JavaScript, donde se llaman automáticamente cuando se asigna (set) o se lee (get) el valor de la propiedad.

Archivo: src/main.ts

```
class Person {
   private _name: string;

   constructor(name: string) {
      this._name = name; // Llamada implícita a setter
   }

   get name() : string { // Devuelve un string
      return this._name;
   }

   set name(name: string) { // Devuelve un  string
      this._name = name;
   }
}

let p: Person = new Person("Peter");
console.log(p.name); // Llamada al  getter implícito
```

Archivo: js/main.js (compiled version)

```
var Person = (function () {
   function Person(name) {
      this._name = name; // llama implícita al setter
   }
   Object.defineProperty(Person.prototype, "name", {
      get: function () {
         return this._name;
      },
      set: function (name) {
         this._name = name;
      },
      enumerable: true,
      configurable: true
   });
   return Person;
}());
```

Casting de tipos

Si tenemos un objeto de tipo **ClassA** (por ejemplo), y sabemos que ese objeto es también una instancia de la clase **ClassB**, la cual hereda de ClassA (un subtipo, vamos), podemos hacer un casting (conversión) al nuevo tipo poniendo **<ClassB>** delante. De una clase derivada a una clase padre, el casting es automático.

```
let table: HTMLTableElement = <HTMLTableElement>document.getElementById("table1"); //  HTMLElement
let elem: HTMLElement = table; // Puede ser asignado directamente  (HTMLElement es una clase padre)
```

Algunos métodos como getElementById, devuelven un objeto del tipo HTMLElement. Es una interfaz que todos los elementos del DOM implementan, pero solo permite el acceso a las propiedades comunes. Si queremos acceder a las propiedades específicas de un elemento, debemos hacer un cast al tipo de objeto específico que sabemos que es (o podemos averiguar usando instanceof).

```
let input: HTMLInputElement = <HTMLInputElement>document.getElementById("input1"); //  HTMLElement
let val: string = input.value; // Ahora podemos acceder a la propiedad “value”

let val2: string = (<HTMLInputElement>document.getElementById("input1")).value; // En un solo paso

let element = document.getElementById("elem1");// HTMLElement

if(element instanceof HTMLImageElement) { // Comprobamos si el objeto es del tipo HTMLImageElement
   console.log((<HTMLImageElement> element).src);  // Hacemos el cast
}
```

Herencia de clases (extends)

La principal diferencia con ES2015 en la herencia es que ahora usamos atributos y métodos privados. Estos atributos y métodos no estarán disponibles en la clase derivada, tal como ocurre en otros lenguajes de programación: Java, C#, …

```
class Vehicle {
   constructor(private name: string) {}

   public setName(name: string) {
      this.name = name;
   }

   public getName(): string {
      return this.name;
   }
}

class Plane extends Vehicle {
   constructor() {
      super("Plane"); // Debemos llamar al constructor del padre siempre
   }

   upgrade() {
      this.name = "Spaceship"; → Property 'name' is private and only accessible within class Vehicle
      this.setName("Spaceship"); // OK, esto sí funciona porque setName es público
   }
}
```

Implementar interfaces en clases

Las interfaces son contratos de código. Se usan para tener consistencia entre las diferentes clases que tienen características en común. Con una interfaz podemos establecer los métodos que debe tener una clase que implementa dicha interfaz.

```
interface IDataBase {
   getData(Json: any): any[];  // Las clases que la implementen deberán tener este método
}

class CarDataBase implements IDataBase {
   getData(json: any): any[] {
      return json.cars;
   }
}

let data: any = {
   people: [ { name: "John", age: 23 }, { name: "Rose", age: 28 } ],
   cars: [ { model: "Tesla", price: 70000 }, { model: "Ford", price: 25000, year: 2015 }]
};

let carDb = new CarDataBase();
console.log(JSON.stringify(carDb.getData(data))); // [{"model":"Tesla","price":70000},
{"model":"Ford","price":25000,"year":2015}]
```

Definir propiedades de objetos con interfaces

Con las interfaces podemos definir qué propiedades (públicas) debe tener un objeto, e incluso propiedades opcionales. Esto es útil cuando trabajamos con objetos JSON por ejemplo, donde podemos definir las propiedades de un objeto.

```
interface ICar {
   model: string;
   price: number;
   year?: number; // Opcional
}
```

```
interface IPerson {
   name: string;
   age: number;
}

interface IAppData {
   people: IPerson[]; // Array de objetos con las propiedades de IPerson
   cars: ICar[]; // Array con las propiedades de ICar
}

interface IData {
   getData(Json: IAppData): Object[];
}

class CarDataBase implements IData {
   getData(json: IAppData): ICar[] { // Recibe un IAppData y devuelve array de ICar
      return json.cars;
   }
}

let data: IAppData = { // Mira la interfaz IAppData
   people: [ { name: "John", age: 23 }, { name: "Rose", age: 28 } ],
   cars: [ { model: "Tesla", price: 70000 }, { model: "Ford", price: 25000, year: 2015 }]
};

let carDb: CarDataBase = new CarDataBase();
let cars: ICar[] = carDb.getData(data);
let car: ICar = cars[0];
console.log(JSON.stringify(car)); // {"model":"Tesla","price":70000}
```

Las interfaces también pueden heredar de otras interfaces:

```
interface IA {
   a: string;
}

interface IB extends IA { // Hereda la propiedad “a” de IA
   b: number;
}

let obj: IB = {a: "hello", b: 5}; // Debería tener las propiedades 'a' y 'b'.
```

Importante: Las interfaces y en general el tipado es muy útil de cara a que el editor (uno que entienda TypeScript como Visual Studio Code) nos autocomplete y sepa qué propiedades (o métodos) y de qué tipo deben aparecer cuando ponga el punto detrás de la variable que referencia al objeto, o al crear un objeto, etc. Esta práctica además evita fallos de que nos equivoquemos en el nombre y en el tipo de propiedades, parámetros, variables, etc.

Otras características de TypeScript

Cuando usamos una librería externa de JavaScript, como JQuery, podemos perder la posibilidad del autocompletado ya que no tendremos acceso a los tipos (no están escritas en TypeScript). Para integrar estas librerías con TypeScript debemos instalar (si están disponibles), los **archivos de definición de tipos (TypeScript)**. Este archivo contendrá principalmente **interfaces** que representan las propiedades, métodos (y sus tipos), de las clases y objetos de dicha librería.

Definición de archivos (TypeScript 2.x)

Desde la versión de TypeScript 2.0.0, se ha vuelto más fácil (e integrarlo con NPM) la instalación de archivos de definición en nuestro proyecto.

Primero, para hacer una definición de búsqueda, podemos usar:

https://microsoft.github.io/TypeSearch/

Una vez que lo has encontrado, lo instalas usando NPM (Como dependencia de desarrollo). Ejemplo: JQuery:

```
npm install --save @types/jquery
```

Ahora tendremos JQuery integrado con el código TypeScript. Todo lo que debes hacer es importar la librería cuando quieras usarla:

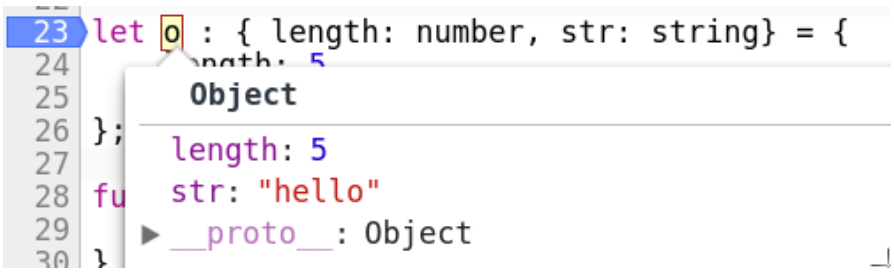
```
import * as $ from "jquery";

$.ajax(....);
```

Depuración en Chrome

Para cada archivo de TypeScript, el compilador genera dos archivos. El archivo traducido de .js y un mapa (.js.map). Este último archivo es bastante útil para referenciar el código TypeScript a partir del .js, y así depurar directamente desde nuestro código TypeScript en las herramientas de desarrollo.

Para poder depurar, abrimos las herramientas de desarrollo y vamos a la pestaña de **sources**. Podemos ver el código TypeScript y establecer puntos de parada, inspeccionar variables, etc. Recarga la página y debería funcionar.



Integrando librerías en TypeScript

Algunas librerías ya incluyen por defecto archivos de definición de tipos para TypeScript por lo que se integrarán directamente sin que necesitemos hacer nada (solo hacer un **import** de lo que necesitemos en nuestro código). Para otras librerías, tendremos que descargar aparte los archivos de definición de tipos como se ha explicado en el punto anterior.

Otras librerías como Handlebars necesitan importar archivos que no contienen JavaScript / TypeScript, como las plantillas, y necesitan ciertos pasos extra para integrarse con TypeScript.

Google Maps

Por defecto, nuestro código TypeScript no reconocerá el objeto global **google** que te crea la librería. Sólo necesitamos descargar el archivo de definiciones y arreglado (no necesitamos importar nada en nuestro código, la variable es global).

```
npm i @types/googlemaps
```

Hay otra opción. Podemos usar una librería que cargue la API de Google Maps e incluya los archivos de definiciones necesarios. Como por ejemplo google-maps-promise que carga la librería de forma asíncrona usando Promesas.

```
npm i google-maps-promise
```

Handlebars

En TypeScript sólo podemos importar de archivos .js o .ts (no podemos importar archivos de plantillas .handlebars por ejemplo). Sólo podemos importarlos usando la sintáxis de NodeJS (**require**). Como TypeScript no reconoce esa función de NodeJS (a no ser que instalemos el archivo de definiciones para Node), usaremos una sintaxis especial de TypeScript: **declare**.

Esta instrucción le indica al compilador que asuma que una variable (o función) existe, indicando el tipo de datos de la misma. Esto es útil para declarar variables globales que TypeScript desconoce, pero que sabemos que existen. En este caso la función require la utilizará Webpack por lo que viene implementada.

```
declare function require(module: string): any;  // Ahora podemos usar require
let template = require('../templates/template.handlebars');
```