

TypeScript

An abstract geometric pattern composed of various shades of blue triangles and diamonds, creating a 3D effect. The pattern is denser at the top and fades out towards the bottom.

Emmanuel Valverde Ramos
Pedro Hernández-Mora de Fuentes

Table of Contents

Introducción	1.1
Instalación del entorno de desarrollo	1.1.1
IDE - Visual Studio Code	1.1.2
Tipos de datos	1.2
Tipos primitivos	1.2.1
Tuple / Tuplas	1.2.2
Enum	1.2.3
Any	1.2.4
Void	1.2.5
Let	1.2.6
Const	1.2.7
For in	1.2.8
For of	1.2.9
Funciones	1.2.10
Genéricos	1.2.11
Assert	1.2.12
Type Alias	1.2.13
Type Union	1.2.14
Type Guards	1.2.15
Fat arrow	1.3
Desestructuración	1.4
Estructuración	1.5
Promesas	1.6
Generators	1.7
Esperas asincrónicas - Async Await	1.8
Clases	1.9
Modificadores de clase	1.9.1
Abstract	1.9.2
IIFE	1.9.3
Herencia	1.9.4

Sobrecarga de metodos	1.9.5
Mixin	1.9.5.1
Interfaces	1.9.6
Decorators	1.10
Class decorator	1.10.1
Property decorator	1.10.2
Method decorator	1.10.3
Static method decorator	1.10.4
Parameter decorator	1.10.5
Módulos	1.11
Sistemas de automatización	1.12
Consejos	1.13
Devolver un objeto literal	1.13.1
Clases estáticas	1.13.2
Métodos de inicialización estáticos	1.13.3
¿Quiénes somos?	1.14

Licencia



Este obra está bajo una [licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/).

Descargas

- [PDF](#)
- [EPUB](#)
- [MOBI](#)
- [URL Base](#)

Descargado en: www.detodoprogramacion.org

Introducción

TypeScript es un lenguaje de programación moderno que permite crear aplicaciones web robustas en JavaScript. TypeScript no requiere de ningún tipo de plugin, puesto que lo que hace es generar código JavaScript que se ejecuta en cualquier navegador, plataforma o sistema operativo.

TypeScript es un "transpilador", es decir, un compilador que se encarga de traducir las instrucciones de un lenguaje a otro, aquí lo llamaremos también pre-compilador ya que este realmente intenta realizar las funciones de un compilador más las funciones de un traductor de instrucciones.

TypeScript es un lenguaje pre-compilado, es decir, un lenguaje el cual será compilado finalmente a javascript, la versión del javascript en la cual será compilado junto con otras configuraciones estará en el archivo **tsconfig**, TypeScript nos proporciona una serie de ventajas sobre javascript, o ES2016,..., ya que tiene una serie de características que ES* no suele tener, como por ejemplo:

- Interfaces
- Clases (Clases de verdad)
- Es furtemente tipado

Aparte de estas características TS tiene más características, pero quizás la más importante sea que gracias a el tiempo de debuggin es reducido ya que, para poder debuggear código javascript en la actualidad, este tiene que ser ejecutado en la aplicación y se le debe sumar el tiempo que se tarde en la detección del fallo, mientras que con TypeScript el código simplemente no será compilado y nos dará un error en la compilación diciéndonos donde se encuentra el error.

En lo referente a las interfaces, estas ni siquiera serán escritas en el código final, simplemente será el mismo transpilador de TypeScript el que se encargará de que el **"Contrato"** sea cumplido, las interfaces en TypeScript pueden ser heredadas esto lo veremos más adelante.

TypeScript es fuertemente tipado ya que requiere de que se le especifiquen los tipos de datos que se quieren utilizar (en caso de que no se especifiquen serán de tipo **ANY**, pero eso lo veremos más adelante).

Instalación del entorno de desarrollo

La instalación de **TypeScript** es relativamente simple, únicamente necesitamos la instalación de un servidor **NodeJS** y con el gestor de paquetes **npm** para descargarnos el transpilador **TypeScript**

Para descargar **NodeJS** hay que ir a nodejs.org

y una vez instalado comprobar la instalación mediante el comando:

```
node -v
```

Si nos dice la versión de NodeJS que tenemos proseguiremos con el siguiente paso la descarga de TypeScript para ello abriremos una terminal y escribiremos esto (Windows/Linux/Mac)

```
npm install -g typescript
```

Para comprobar la correcta instalación de TypeScript y la versión que se a instalado escribimos

```
tsc -v
```

instalación de los tsd de nodejs

```
npm install tsd -g
```

tsc = TypeScript Console

El siguiente paso será crear una carpeta donde trabajar para ello dependiendo de nuestro sistema operativo podremos utilizar un ejecutable u otro para crear una carpeta o simplemente hacerlo desde la interfaz gráfica, una vez creada navegaremos a través de la terminal a la carpeta recién creada y escribiremos el siguiente comando

```
tsc --init
```

Con este comando generaremos el archivo de configuración básico que utilizará TypeScript para compilar la información.

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "noImplicitAny": false,  
    "sourceMap": false  
  },  
  "exclude": [  
    "node_modules"  
  ]  
}
```

La presencia de este archivo significa que este directorio es la raíz del proyecto.

Para que se compile un fichero TypeScript tienes que utilizar el siguiente comando

```
tsc -w
```

Si al realizar esta instalación ocurre algún error, una alternativa para practicar sería [la página de pruebas de TypeScript](#)

IDE - Visual Studio Code

<https://code.visualstudio.com/> En Visual Code es un entorno de desarrolladores de **Microsoft**, pero de código abierto, la razón por la que aparece este entorno de desarrollo, es porque ni más ni menos que **Microsoft** es el creador de **Typescript**, VS Code es un IDE que nos permitirá automatizar tareas para ello abrimos el menú pulsando el `F1` y escribimos `Configure task Runner` hacemos click y seleccionamos `tsconfig.json` esto nos generará una carpeta llamada `tasks` y dentro de ella un archivo llamado `task.json` que sería así:

```
{

  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format

  "version": "0.1.0",
  "command": "tsc",
  "isShellCommand": true,
  "args": ["-p", "."],
  "showOutput": "silent",
  "problemMatcher": "$tsc"
}
```

Buscador

Para buscar en todos los ficheros de un proyecto utilizamos `ctrl + alt + t`

Debuggin

Para poder debuggear en Visual Code ts, necesitamos configurar el archivo de configuración `tsconfig` y en la línea `sourceMap` la ponemos a `true` dejando el fichero de configuración de la siguiente forma:


```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "noImplicitAny": false,  
    "sourceMap": true  
  },  
  "exclude": [  
    "node_modules"  
  ]  
}
```

Una vez modificada esa línea se generará un archivo .map con el mismo nombre del archivo .ts que tengamos en ese archivo .map estarán asociadas las líneas del archivo .js compilado y la línea del archivo .ts una vez modificado, pondremos el punto de debug donde deseemos y presionaremos **F5** para ejecutar el código aparecerá un dialogo que nos preguntará que tipo de servidor queremos elegir, elegiremos **NodeJS** y configuraremos el archivo de configuración para que quede así:

```
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "name": "Iniciar",  
      "type": "node",  
      "request": "launch",  
      "program": "${workspaceRoot}/app.ts",  
      "stopOnEntry": false,  
      "args": [],  
    }  
  ]  
}
```

```
"cwd": "${workspaceRoot}",

"preLaunchTask": null,

"runtimeExecutable": null,

"runtimeArgs": [

    "--no-lazy"

],

"env": {

    "NODE_ENV": "development"

},

"externalConsole": false,

"sourceMaps": true,

"outDir": null

},

{

    "name": "Asociar",

    "type": "node",

    "request": "attach",

    "port": 5858,

    "address": "localhost",

    "restart": false,

    "sourceMaps": false,

    "outDir": null,

    "localRoot": "${workspaceRoot}",

    "remoteRoot": null

},

{

    "name": "Attach to Process",
```

```
        "type": "node",  
  
        "request": "attach",  
  
        "processId": "${command.PickProcess}",  
  
        "port": 5858,  
  
        "sourceMaps": false,  
  
        "outDir": null  
    }  
]  
}
```

Las líneas modificadas del original han sido las siguientes:

```
"sourceMaps": true,  
  
"program": "${workspaceRoot}/app.ts",
```

Para debuggear al siguiente punto de debug utilizamos el botón **F10** y para un paso más controlado un **F11**

Tipos de datos

TypeScript es un lenguaje que añade a **JavaScript** una capa de tipado estático y algunas otras incorporaciones de **OOP tradicional**. Esta capa puede resultarnos de muchísima ayuda durante el desarrollo. Sin embargo, todas estas características son simplemente para ayudar a trabajar con JavaScript en tiempo de diseño, ya que **TypeScript** compila todo como JavaScript tradicional.

Tipado estático o fuertemente tipado: Se debe de definir el tipo de dato, obligando a que no pueda haber errores con los tipos de datos

Tipado dinámico o débilmente tipado: No se deben de o tiene porque especificar el tipo de dato (PHP, Javascript)

Porque **typescript** es fuertemente tipado

```
var a=3;

var b="hola";

var c=a+b; // Resultado 3hola
```

```
if ("0" == 0) // es true

if ("3" === 3) // es false
```

Estos ejemplos son posibles problemas que tienen los lenguajes débilmente tipados

Tipos primitivos

Boolean

true o false

```
let isDone: boolean = false;
```

Number

Datos numéricos

```
let decimal: number = 6;

let hex: number = 0xf00d;

let binary: number = 0b1010;

let octal: number = 0o744
```

String

Cadenas de caracteres y/o textos

```
let color: string = "blue"; //
color = 'red';
```

También se pueden utilizar *"Templates"* plantillas para concatenar strings como por ejemplo:

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${fullName}. I'll be ${age + 1} years old next month.`
```

Para poder utilizar esta sintaxis los string deben estar contenidos entre ```.

Este tipo de sintaxis es el equivalente a:

```
let sentence: string = "Hello, my name is " + fullName + "." + "I'll be " + (age + 1) + " years old next month."
```

Plantillas de strings

Las plantillas de strings se escriben entre ``` y la sintaxis sería:

```
var lyrics = 'Never gonna give you up'; // entre comillas simples
var html = `

${lyrics}</div>`; // entre tilde inversa


```

Este tipo de plantillas permite que podamos utilizar más de una línea sin tener que utilizar el operador `+` por eso se dice que un "string templated" es **multilineal**

String literal type

Podemos crear un string literal como un type. Por ejemplo:

```
let literalString = 'Hello';
```

La variable que acabamos de crear nada más que podrá contener el valor que le hemos asignado, es decir 'Hola'.

```
let literalString = 'Hello';
literalString = 'Bye'; // Error: "Bye" is not assignable to type "Hello"
```

Por si solo no tiene una gran utilidad por lo que se combina con [union types](#), [type guards](#), y [type alias](#). Los cuales explicaremos más tarde.

```
type CardinalDirection =
  "North"
  | "East"
  | "South"
  | "West";

function move(distance: number, direction: CardinalDirection) {
  // ...
}

move(1, "North"); // Okay
move(1, "Nurth"); // Error!
```

Array

Arrays, sino se les especifica tipo son **ANY**

```
let list: number[] = [1, 2, 3];
```

Con esta sintaxis se puede especificar qué tipo de datos debe haber en el array

```
let list: Array<number> = [1, 2, 3];
```

Null

Es cuando un objeto o variable no esta accesible.

Undefined

Es cuando un objeto o variabe existe pero no tiene un valor. Si nuestro código interactua con alguna API podemos recibir null como respuesta, para evaluar esas respuestas es mejor utilizar `==` en vez de `===`

```
// ----- ejemplo.ts -----  
console.log(undefined == undefined); // true  
console.log(null == undefined); // true  
console.log(0 == undefined); // false  
console.log('' == undefined); // false  
console.log(false == undefined); // false
```

Tuple / Tuplas

Como en base de datos, hacen referencia a registros clave / valor

```
// Declaración de tublas

let x: [string, number];

// Inicialización correcta

x = ["hello", 10]; // OK

// Inicialización incorrecta

x = [10, "hello"]; // Error
```

Para acceder a los datos dentro de las tuplas de las cuales sabes el índice se hace así:

```
console.log(x[0].substr(1)); // OK

console.log(x[1].substr(1)); // Error, Un tipo 'number' no tiene la función 'substr'
```

Cuando queramos acceder a un elemento sin conocer el mapeo del contenido

```
x[3] = "world"; // OK, Un tipo 'string' puede ser asignado a una tupla que contenga 'string | number'

console.log(x[5].toString()); // OK, Un tipo 'string' y un tipo 'number' tienen la función 'toString'

x[6] = true; // Error, El tipo 'boolean' no es 'string | number'
```


Enum

Los enumerados en **TypeScript**, son distintos a los enumerados de otros lenguajes de programación, estos solo almacenan números para identificar las constantes.

Si no se le especifica el valor por defecto se lo asigna normalmente, también es importante saber, que los enumerados no aceptan que su valor sea un String, solamente número

```
enum Direction {  
  
    Up = 1, // Si se le asigna un valor numerico primero, los siguientes valores empi  
ezan desde el número especificado  
    Down = NaN, // Si le ponemos un NaN después de haber inicializado un valor nos ob  
liga a inicializar el siguiente después de este, esto no solo pasa con Nan, pasa con St  
ring.length, etc.  
    Left = "asdasd".length,  
    Right = 1 << 1 // También admiten operadores binarios  
}  
  
var a = Direction.Up;  
  
console.log(Direction.Down);
```

Sin asignación de valor

```
enum Color {Red, Green, Blue};  
let c: Color = Color.Green; // 1
```

Con asignación de valor

```
enum Color {Red = 1, Green = 2, Blue = 4};  
let c: Color = Color.Green; // 2
```

También se puede acceder al nombre de los atributos

```
enum Color {Red = 1, Green, Blue};  
  
let colorName: string = Color[2];  
  
alert(colorName); // Green
```

Es muy importante saber que distintos enumerados no pueden ser comparados ya que el nombre de los enumerados no es el mismo, aunque puedan tener el mismo índice numérico.

Para comprobar lo que digo utilizaré [Type alias](#), y el ejemplo es el siguiente:

```
// Foo
enum FooIdBrand {}
type FooId = FooIdBrand & string;

// Bar
enum BarIdBrand {}
type BarId = BarIdBrand & string;

/**
 * Demo
 */
var fooId: FooId;
var barId: BarId;

// Por seguridad
fooId = barId; // error
barId = fooId; // error

// Newing up
fooId = 'foo' as FooId;
barId = 'bar' as BarId;

// Los dos tipos son compatibles con la base
// que en este caso es string
var str: string;
str = fooId;
str = barId;
```

Any

Puede ser cualquier tipo de objeto de javascript

```
let notSure: any = 4;
notSure = "maybe a string instead"; // typeof = string
notSure = false;; // typeof = boolean
```

```
let notSure: any = 4;
notSure.ifItExists(); // OK, ifItExists puede existir
notSure.toFixed(); // OK, toFixed existe, pero no es comprobado por el compilador
let prettySure: Object = 4;
prettySure.toFixed(); // Error: La propiedad 'toFixed' no existe en un 'Object'.
```

```
let list: any[] = [1, true, "free"];
list[1] = 100;
```

Void

```
function warnUser(): void {  
    alert("This is my warning message");  
}
```

Este tipo de dato no es recomendable para variables ya que solo pueden ser asignados valores null o undefined

```
let unusable: void = undefined;
```

Let

En javascript hay dos formas de declarar variables: `var` y `let`, `var` no tiene un ámbito de bloque mientras que `let` sí.

`var`

```
var foo = 123;
if (true) {
  var foo = 456;
}
console.log(foo); // 456
```

`let`

```
let foo = 123;
if (true) {
  let foo = 456;
}
console.log(foo); // 123
```

Const

Ha sido añadido en ES6 / TypeScript permitiendonos añadir variables inmutables también conocidas como constantes. El uso de `const` es una buena práctica de mantenimiento y legibilidad. **Las constantes deben ser declaradas y asignadas siempre.**

```
const foo = 123;  
foo = 456; // NO permitido
```

Las constantes también admiten objetos literales como por ejemplo:

```
const foo = { bar: 123 };  
foo = { bar: 456 }; // ERROR no se permite la modificación de objeto
```

Pero si se puede modificar el contenido de las variables que contiene el objeto literal, ejemplo:

```
const foo = { bar: 123 };  
foo.bar = 456; // Permitido  
console.log(foo); // { bar: 456 }
```

For in

For in es una característica que ya tenía javascript ,y no ha sido mejorada en TypeScript, mediante la cual puedes acceder y recorrer objetos y arrays y obtener tanto los índices como los valores.

For in accediendo al valor de una variable dentro de un objeto:

TypeScript

```
let list = {a: 1, b: 2, c:3};

for(let i in list){
  console.log(list[i]); // 1, 2, 3
}
```

Javascript

```
var list = { a: 1, b: 2, c: 3 };

for (var i in list) {
  console.log(list[i]); // 1, 2, 3
}
```

For in accediendo al índice de una variable dentro de un objeto;

TypeScript

```
let list = {a: 1, b: 2, c:3};

for(let i in list){
  console.log(i); // a, b, c
}
```

Javascript

```
var list = { a: 1, b: 2, c: 3 };

for (var i in list) {
  console.log(i); // a, b, c
}
```


For of

For of es una característica nueva de ES6 con la **cual puedes acceder y recorrer arrays y strings obteniendo su valor**, es decir, no puede recorrer objetos. Aunque se podrían recorrer objetos en el caso de que estos fueran creados por clases que implementen `Symbol.iterator`. `for ... of` también tiene un peor rendimiento en comparación con el `for...in` ya que al compilarlo a JS crea más variables y hace más comprobaciones.

For of accediendo al valor de una variable dentro de un array:

TypeScript

```
let list = ["a", "b", "c"];

for (let b of list) {
    console.log(b); // a, b, c
}
```

Javascript

```
var list = ["a", "b", "c"];

for (var _i = 0, list_1 = list; _i < list_1.length; _i++) {
    var b = list_1[_i];
    console.log(b); // a, b, c
}
```

For of accediendo al valor de una variable dentro de un string:

TypeScript

```
let string = "is it me you're looking for?";

for (let char of string) {
    console.log(char); // is it me you're looking for?
}
```

Javascript

```
var string = "is it me you're looking for?";

for (var _i = 0, string_1 = string; _i < string_1.length; _i++) {
    var char = string_1[_i];
    console.log(char); // is it me you're looking for?
}
```

For of accediendo al valor de una variable dentro de un objeto, el cual nos dará error:

TypeScript

```
let obj = {a: 1, b: 2, c:3};

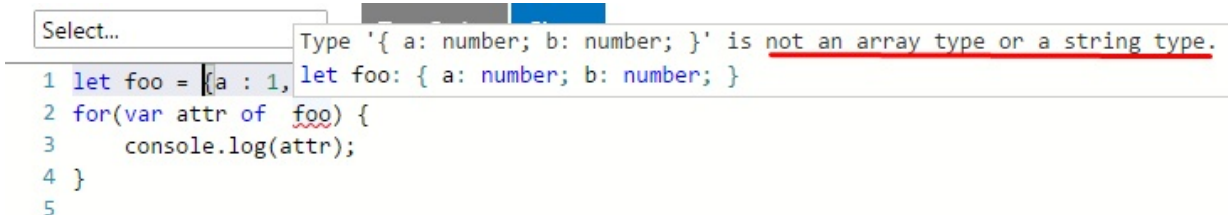
for(let i of obj){
    console.log(i); // Error
}
```

Javascript

```
var obj = { a: 1, b: 2, c: 3 };

for (var _i = 0, obj_1 = obj; _i < obj_1.length; _i++) {
    var i = obj_1[_i];
    console.log(i); // Error
}
```

El error según el transpilador de **TypeScript** Es el siguiente:



```
1 let foo = {a: 1, b: 2};
2 for(var attr of foo) {
3     console.log(attr);
4 }
5
```

Type '{ a: number; b: number; }' is not an array type or a string type.

Funciones

Este tipo de función hace referencia al objeto que llama a esta función

```
setTimeout(function(){  
  
    console.log(this); // Elemento Que llama a la función  
  
}, 2000);
```

Este tipo de funciones, lo que hacen es que el this no hace referencia al padre sino al objeto que contiene la función

```
setTimeout(() => {  
  
    console.log(this); // Elemento que contiene esta función  
  
}, 2000);
```

Ejemplos sobre como evitar el tipo Any y filtrar solo por los tipos de datos que necesitamos

```
// para poder definir tipos utilizaremos el 0 lógico  
  
function padLeft(value: string, padding: string | number) {  
  
    if(typeof padding === "number"){  
        return Array(padding + 1).join(" ") + value;  
    }  
  
    if(typeof padding === "string") {  
        return Array(padding.length + 1).join(" ") + value;  
    }  
  
    // Si existiera Any tendríamos que controlar la excepción  
    // Como buenas practicas y ya que este código al fin y al cabo  
    // será javascript y es vulnerable a inyección siempre está bien  
    // realizar el control de las posibles excepciones  
    throw new Error(`Expected String or number, got '${padding}'`);  
  
}  
  
console.log(padLeft("hello", "aaa")); // Ejemplo de función con texto -> Funciona  
console.log(padLeft("hello", 5)); // Ejemplo de función con número -> Funciona  
console.log(padLeft("hello", true)); // Ejemplo de función con texto -> NO FUNCIONA  
A (no compila)
```

TypeScript admite que se declaren parametros opcionales de la siguiente forma utilizando la `?`:

```
// Compiled with --strictNullChecks
function validateEntity(e: Entity?) {
    // Throw exception if e is null or invalid entity
}

function processEntity(e: Entity?) {
    validateEntity(e);
    let s = e!.name; // Assert that e is non-null and access name
}
```

Igualación de funciones

En javascript las funciones pueden ser igualadas, **TypeScript** junto con su nueva sintáxis también permite este comportamiento como por ejemplo, utilizando las **fat arrow**:

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

```
let x = () => ({name: 'Alice'});
let y = () => ({name: 'Alice', location: 'Seattle'});

x = y; // OK
y = x; // Error porque x()nmo tiene la propiedad location
```

Genéricos

Los tipos genéricos, son aquellos que como las interfaces no se verán compilados en Javascript ya que solo están accesibles en tiempo de compilación, La manera adecuada de realizar la sobrecarga de métodos es con los tipos genéricos un ejemplo sería así:

Versión TypeScript

```
function echo<T>(arg: T) : T {  
    return arg;  
}  
  
let a = echo<number>(1); // El typeof es Number  
let b = echo<String>("Hola mundo"); // El typeof es String
```

Versión Javascript (Ya compilado)

```
function echo(arg) {  
    return arg;  
}  
  
var a = echo(1); // El typeof es Number  
var b = echo("Hola mundo"); // El typeof es String
```

La diferencia entre esta forma y la otra, es que de esta forma, podríamos recibir cualquier tipo de objeto, y no deberíamos especificar el tipo de objeto que esperamos, esto esta muy bien ya que está diseñado para los objetos que no son primitivos de javascript. Con esto evitamos el **Any** y mejoraría la manera de realizar la sobrecarga (Lejos de como sería en Java o C#).

Con los tipos genéricos se debe tener cuidado, ya que no todos los métodos estan disponibles para todos lo tipos de objetos.

TypeScript

```
class Generic<T> {  
    add: (X: T, y:T) => T;  
}  
  
let myGeneric = new Generic<number>();  
  
console.log(myGeneric.add = function (x,y) {return x + y});  
console.log(myGeneric.add(3,4));
```

Javascript

```
var Generic = (function () {

    function Generic() {

    }

    return Generic;

})();

var myGeneric = new Generic();
console.log(myGeneric.add = function (x, y) { return x + y; });
console.log(myGeneric.add(3, 4));
```

Como se puede apreciar en este ejemplo podemos declarar una función dentro de una clase que devolverá lo que le pasemos por parametro, permitiendonos así modificar los returns de nuestras funciones según queramos.

Para poder pasar como parametro a una función y asegurarnos de que ese parámetro tiene un método en concreto deberemos implementar una interfaz y forzar al parámetro que se le pasará a la función a utilizar dicha interfaz.

TypeScript Nota: Es muy importante que veáis que cuando se implementa una interfaz en un parámetro utilizamos la palabra reservada **extends** y no la palabra reservada **implements**

```
// Interfaz que asegura que el parametro tenga el metodo length
interface withLength {
    length: number;
}

// El parametro hereda de la interfaz la cual fuerza al parametro tenga el método length
function echo<T extends withLength>(arg: T): T {
    console.log(arg.length);
    return arg;
}

// Esto funcionará
let a = echo("aaa");
let t = echo({length: 2, name: "aa"});

// Esto NO funcionará
let b = echo(1);
```

Javascript

```
// El parametro hereda de la interfaz la cual fuerza al parametro tenga el método length
function echo(arg) {
    console.log(arg.length);
    return arg;
}

// Esto funcionará
var a = echo("aaa");
var t = echo({ length: 2, name: "aa" });

// Esto NO funcionará
var b = echo(1);
```

También podemos hacer que los atributos que intentamos modificar se encuentren dentro del tipo de objeto que le pasa, eso sería de la siguiente forma.

```
function copyFields<T extends U, U>(source: T, target: U) : U {

    for(let id in source){

        if(target[id] != undefined){

            source[id] = target[id];

        }else {

            target[id] = source[id];

        }

    }

    return target;
}

let a = {a: 1, b: 2, c: 3};

let b = copyFields (a, {b: 10, c:20}); // Esto funcionará
let c = copyFields (a, {Q: 20}); // Esto NO funcionará

console.log(b); // 1, 10, 20
console.log(c); // Lo devuelve si lo compilas a pesar de saber que está mal
```

Aserción de tipos (Assert)

```
class Bird {
    fly(){
        console.log("Pajaro");
    }

    layEggs(){
        console.log("Pone huevos");
    }
}

class Fish {

    constructor(){
        // Solo para el ejemplo
    }

    swim(){
        console.log("PEZ")
    }

    layEggs(){
        console.log("Pone huevos");
    }
}

function getAnimal() : Fish | Bird {
    var a : Fish = new Fish();
    return a;
}

let pet = getAnimal();
console.log(getAnimal());
pet.layEggs();

// ASERCIÓN
if((<Fish>pet).swim){
    (<Fish>pet).swim();
} else if((<Bird>pet).fly) {
    (<Bird>pet).fly();
}
```

Y si ahora añadieramos una clase más este condicional no controlaria esa situación, en este ejemplo al ser tan simple se ve claro, pero en clases más complejas con lógicas más complejas puede llegar ha ser un gran problema.

Type Alias

Los **Type Alias** son exactamente los mismos tipos de datos y valores originales solo que con nombres alternativos, esto sirve para darle más semántica al lenguaje.

Type alias utiliza la palabra reservada `type` para funcionar:

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === "string") {
        return n;
    }
    else {
        return n();
    }
}
```

los **Type alias** no solo pueden ser tipos de datos genéricos, también se pueden utilizar parametros como por ejemplo:

```
type Container<T> = { value: T };
```

También puede hacer referencia a sí mismo:

```
type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

El operador `&` se utiliza para crear un tipo de dato base como por ejemplo:

```
// F00
enum FooIdBrand {}
type FooId = FooIdBrand & string;

// BAR
enum BarIdBrand {}
type BarId = BarIdBrand & string;
```


Type Union

Implica que solo los métodos que sean iguales de ambas interfaces o clases (solamente el nombre del método sean iguales, el contenido puede ser distinto) podrán ser utilizados allí donde se utilicen métodos de unión como el siguiente ejemplo:

```
interface Bird {
    fly();
    layEggs(); // Los 2 pueden
}

interface Fish {
    swim();
    layEggs(); // Los 2 pueden
}

function getAnimal() : Fish | Bird {
    var a : Fish;
    return a;
}

let pet = getAnimal();
pet.layEggs(); // Esto funcionaría
pet.swim() // Esto da error
```

Type Guards

Son las maneras de controlar los tipos de datos\objetos que se están utilizando, esto rompe con la programación orientada a objetos ya que esto representa un problema para el polimorfismo, por ejemplo: si estuviéramos haciendo una serie de comprobaciones para que según el tipo de clase u objeto se realice una acción u otra en el momento en el que existiera un objeto o clase que no tuviéramos contemplado tendríamos que modificar todo el código.

```
interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getAnimal() : Fish | Bird {
    var a : Fish;
    return a;
}

function isFish(pet: Fish | Bird): pet is Fish {
    return (<Fish>pet).swim != undefined;
}

let pet = getAnimal();

if(isFish(pet)) {
    (<Fish>pet).swim();
    console.log('glup');
} else {
    console.log('believe i can fly');
    pet.fly();
}
```

Fat arrow

Las funciones **fat arrow** se utilizan en para:

- Omitir la palabra `function`

```
var inc = (x)=>x+1;
```

- Para capturar el `this`, ya que en javascript se pierde muy rapidamente el contexto de `this` como por ejemplo en la siguiente situación:

```
function Person(age) {  
  this.age = age  
  // ejecutandola en el navegador el this es window  
  // ya que es quien hace la llamada  
  this.growOld = function() {  
    this.age++;  
  }  
}  
  
var person = new Person(1);  
setTimeout(person.growOld,1000); // debería incrementar 1 + 1 = 2  
setTimeout(function() { console.log(person.age); },2000); // Devuelve 1, debería ser 2
```

Esto es debido a que el `this` que ejecuta la función `growOld` en javascript es `window` y no `Person`. Si utilizáramos una función con fat arrow funciona, esta funcionaría.

```
function Person(age) {  
  this.age = age  
  // aquí el this es el objeto y no quien hace la llamada  
  this.growOld = () => {  
    this.age++;  
  }  
}  
  
var person = new Person(1);  
setTimeout(person.growOld,1000);  
setTimeout(function() { console.log(person.age); },2000); // devuelve 2
```

Otra solución sería mezclar las 2 sintaxis

```
class Person {  
  constructor(public age:number) {}  
  growOld = () => {  
    this.age++;  
  }  
}  
  
var person = new Person(1);  
setTimeout(person.growOld, 1000);  
setTimeout(function() { console.log(person.age); }, 2000); // 2
```

- Para capturar argumentos

¿Cuándo es necesario utilizar una fat arrow?

Es necesario utilizarla cuando la función va a ser llamada por otra clase o por otro método de la siguiente forma:

```
// Se iguala la función de la clase a una variable de otra clase  
var growOld = person.growOld;  
// más adelante se llama  
growOld();
```

Como utilizarlas con librerías que utilizan `this`

Existen muchas librerías como por ejemplo jQuery, para trabajar con este tipo de librerías utilizamos variables auxiliares como por ejemplo `_self`

```
let _self = this;  
something.each(function() {  
  console.log(_self); // the lexically scoped value  
  console.log(this); // the library passed value  
});
```

¿Cómo utilizar funciones con herencia?

En caso de que quieras sobrescribir una función de la clase padre siempre deberemos realizar una copia de este, por ejemplo:

```
class Adder {  
  // This function is now safe to pass around  
  add = (b: string): string => {  
    return this.a + b;  
  }  
}  
  
class ExtendedAdder extends Adder {  
  // Create a copy of parent before creating our own  
  private superAdd = this.add;  
  // Now create our override  
  add = (b: string): string => {  
    return this.superAdd(b);  
  }  
}
```

Cadenas de fat arrow (Currying)

Es una serie de funciones encadenadas el uso es simple y es el siguiente:

```
// A curried function  
let add = (x: number) => (y: number) => x + y;  
  
// Simple usage  
add(123)(456);  
  
// partially applied  
let add123 = add(123);  
  
// fully apply the function  
add123(456);
```


Desestructuración

La desestructuración nos permite extraer valores almacenados en arrays u objetos.

Desestructuración de objetos

```
var obj = {x: 1, y: 2, z: 3};  
console.log(obj.x); // 1  
  
var {x, y, z} = obj;  
console.log(x); // 1
```

Desestructuración de arrays

```
var array = [1, 2, 3];  
console.log(array[0]); // 1  
  
var [x, y, z] = array;  
console.log(x); // 1
```

Desestructuración de arrays con estructuración

```
var array = [1, 2, 3, 4];  
var [x, y, ...rest] = array;  
console.log(x, y, rest); // 1, 2, [3,4]
```

Estructuración

La estructuración de parámetros es una forma rápida de que por ejemplo una función acepte una gran cantidad de parámetros como array.

```
function rest(first, second, ...allOthers) {  
  console.log(allOthers);  
}  
  
rest('foo', 'bar'); // []  
rest('foo', 'bar', 'bas', 'qux'); // ['bas', 'qux']
```

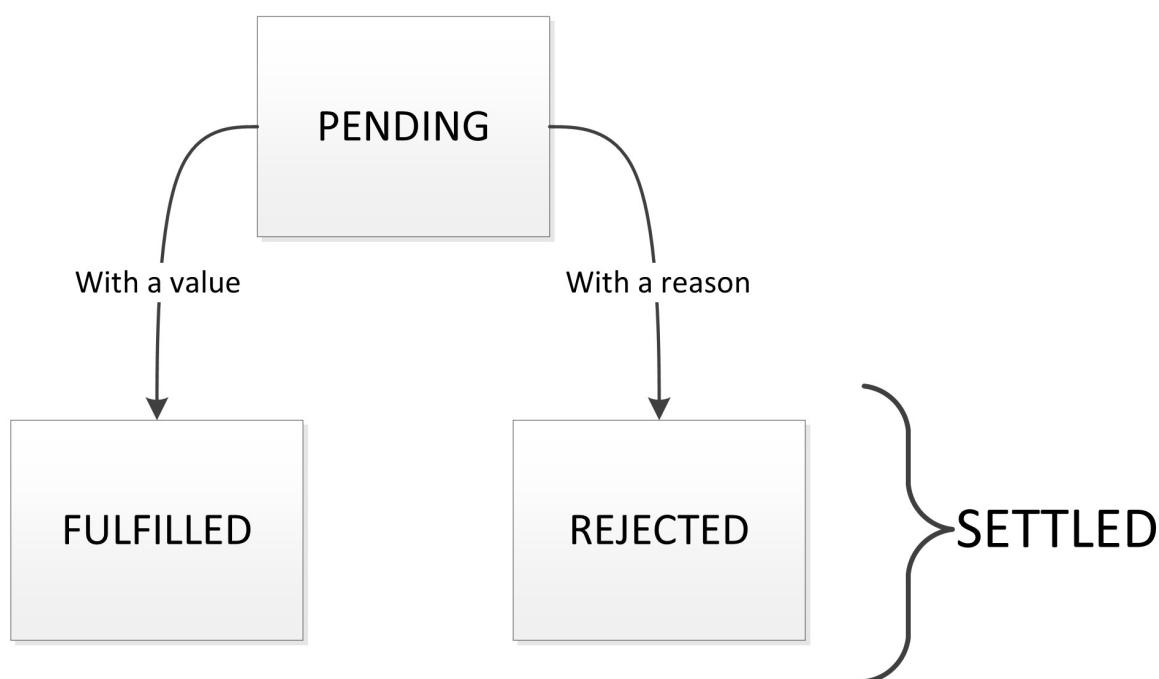
También se puede hacer asignaciones con array, como por ejemplo: ```ts var list = [1, 2]; list = [...list, 3, 4]; console.log(list); // [1,2,3,4]`

Promesas

Las promesas representa un resultado eventual de una operación asíncrona, la primera manera de interactuar con un una promesa o `promise` es a través del método `then` el cual registra el callback que recibirá la respuesta o la razón por la cual la promesa no a podido ser cumplida.

Estados

Las promesas pueden estar en 3 estados `pending` , `fulfilled` y `rejected`



Pending

Cuando una promesa no se haya terminado aún pero aún no ha sido rechazada. (a la espera), y la respuesta podrá ser `fulfilled` o `rejected` .

Fulfilled

Cuando la respuesta ha sido devuelta y procesada correctamente, no podrá cambiar de estado y el valor no debe cambiar (debido a la promesa, por otro procesamiento sí).

Rejected

Cuando ha ocurrido un error en la promesa, el estado de la transición no debe cambiar y debe tener una razón por la cual a sucedido el error la cual no debe cambiar. para obtener los resultados `rejected` utilizamos la palabra reservada `catch`

Cuando digo que algo no debe cambiar quiero decir que deberá ser comprobado con `===`.

[más información sobre promesas](#)

Ejemplos

Ejemplo de llamada a `then`

```
const promise = new Promise((resolve, reject) => {
  resolve(123);
});
promise.then((res) => {
  console.log('I get called:', res === 123); // Devuelve: true
});
promise.catch((err) => {
  // Nunca es utilizado
});
```

Ejemplo de llamada a `catch`

```
const promise = new Promise((resolve, reject) => {
  reject(new Error("Algo malo a pasado"));
});
promise.then((res) => {
  // This is never called
});
promise.catch((err) => {
  console.log('Tengo una llamada: ', err.message); // Tengo una llamada: 'Algo malo a pasado'
});
```

Cadenas de promesas o Chain-ability of Promises

Una cadena de promesas es una manera muy útil de realizar peticiones asíncronas.

Sí una promesa hace un `return` la cadena hace la siguiente petición al `then`

```
Promise.resolve(123)
  .then((res) => {
    console.log(res); // 123
    return 456;
  })
  .then((res) => {
    console.log(res); // 456
    return Promise.resolve(123);
  })
  .then((res) => {
    console.log(res); // 123 : Notice that this `this` is called with the resolved
value
    return Promise.resolve(123);
  })
```

Puedes manejar los errores añadiéndole un método `catch` a la cadena.

```
Promise.reject(new Error('something bad happened'))
  .then((res) => {
    console.log(res); // not called
    return 456;
  })
  .then((res) => {
    console.log(res); // not called
    return Promise.resolve(123);
  })
  .then((res) => {
    console.log(res); // not called
    return Promise.resolve(123);
  })
  .catch((err) => {
    console.log(err.message); // something bad happened
  });
```

También se puede hacer que un método `catch` continúe con la cadena de promesas, de la siguiente manera:



El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la Programación Java, C/C++/C#, Visual.Net, SQL, Python, Javascript, Oracle, Algoritmos, CSS, Desarrollo Web, Joomla, jquery, Ajax y Mucho Mas...

VISITA

www.detodoprogramacion.org

www.detodopython.com

www.gratiscodigo.com



GRATISCODIGO.COM

Usa, lo que ya se creó...



Detodo
Python.com

```
Promise.reject(new Error('something bad happened'))
  .then((res) => {
    console.log(res); // not called
    return 456;
  })
  .catch((err) => {
    console.log(err.message); // something bad happened
    return Promise.resolve(123);
  })
  .then((res) => {
    console.log(res); // 123
  })
```

Cualquier error ocurrido en un `then` llamará al método `catch`. Ej.:

```
Promise.resolve(123)
  .then((res) => {
    throw new Error('something bad happened')
    return 456;
  })
  .then((res) => {
    console.log(res); // never called
    return Promise.resolve(789);
  })
  .catch((err) => {
    console.log(err.message); // something bad happened
  })
```

El hecho de que el primer `then` al dar un error se salte el siguiente `then` siendo una llamada asincrónica, nos provee con un nuevo paradigma el cual nos permite capturar mejor las excepciones asíncronas.

También es posible que algunas funciones puedan devolver promesas como por ejemplo:

```
function iReturnPromiseAfter1Second():Promise<string> {
  return new Promise((resolve)=>{
    setTimeout(()=>resolve("Hello world!"), 1000);
  });
}

Promise.resolve(123)
  .then((res)=>{
    // res is inferred to be of type `number`
    return iReturnPromiseAfter1Second();
  })
  .then((res) => {
    // res is inferred to be of type `string`
    console.log(res); // Hello world!
  });
```

En el siguiente ejemplo veremos como se hace la carga de un `json` de forma asíncrona:

```
// good json file
loadJSONAsync('good.json')
  .then(function (val) { console.log(val); })
  .catch(function (err) {
    console.log('good.json error', err.message); // never called
  })

// non-existent json file
  .then(function () {
    return loadJSONAsync('absent.json');
  })
  .then(function (val) { console.log(val); }) // never called
  .catch(function (err) {
    console.log('absent.json error', err.message);
  })

// invalid json file
  .then(function () {
    return loadJSONAsync('bad.json');
  })
  .then(function (val) { console.log(val); }) // never called
  .catch(function (err) {
    console.log('bad.json error', err.message);
  });
```

Promesas en paralelo

Como habréis observado previamente, hasta ahora todos los ejemplos que hemos visto heren peticiones en serie pero que sentido tiene eso si lo que queremos es una carga asincrónica de la información, pues no mucha es por eso que ahora veremos un ejemplo de como serían las peticiones en paralelo.

TypeScript

```
//----- main.ts -----  
// Una función asincróna simulando la petición desde el servidor  
function loadItem(id: number): Promise<{id: number}> {  
    return new Promise((resolve)=>{  
        console.log('loading item', id);  
        setTimeout(() => { // simulate a server delay  
            resolve({ id: id });  
        }, 1000);  
    });  
}  
  
// Cadena (serie)  
let item1, item2;  
loadItem(1)  
    .then((res) => {  
        item1 = res;  
        return loadItem(2);  
    })  
    .then((res) => {  
        item2 = res;  
        console.log('done');  
    }); // overall time will be around 2s  
  
// Paralelo  
Promise.all([loadItem(1),loadItem(2)])  
    .then((res) => {  
        [item1,item2] = res;  
        console.log('done')  
    }); // overall time will be around 1s
```

Javascript

```
//----- main.js -----  
// Una función asíncrona simulando la petición desde el servidor  
function loadItem(id) {  
    return new Promise((resolve) => {  
        console.log('loading item', id);  
        setTimeout(() => {  
            resolve({ id: id });  
        }, 1000);  
    });  
}  
// Cadena (serie)  
let item1, item2;  
loadItem(1)  
    .then((res) => {  
        item1 = res;  
        return loadItem(2);  
    })  
    .then((res) => {  
        item2 = res;  
        console.log('done');  
    }); // overall time will be around 2s  
// Paralelo  
Promise.all([loadItem(1), loadItem(2)])  
    .then((res) => {  
        [item1, item2] = res;  
        console.log('done');  
    }); // overall time will be around 1s
```

Para poder testear esto ejecutaremos el javascript en nuestro servidor node

```
node main.js
```

[Ejemplo en el playground](#)

Generators

Los `generators` no pueden ser utilizados en `es5` deben ser usados en `es6` o superior.

Los generadores son funciones de las que se puede salir y volver a entrar. Su contexto (asociación de variables) será conservado entre las reentradas. Las reentradas son efectuadas gracias a la palabra reservada `yield`

Los generadores son funciones que devuelven un `generator object`, hay dos grandes motivaciones para utilizar este tipo de funciones, una de ellas es que los `generators` siguen la interfaz `iterator` permitiendo así utilizar los siguientes métodos `next`, `return` y `throw`. Y la otra razón la veremos un poco más adelante.

```
function* infiniteSequence() {
  var i = 0;
  while(true) {
    yield i++;
  }
}

var iterator = infiniteSequence();
while (true) {
  console.log(iterator.next()); // { value: xxxx, done: false } para siempre
}
```

Como se puede apreciar este tipo de funciones devuelve junto con el resultado el estado de la iteración, si la iteración termina bien devuelve `false` sino `true`. Aunque eso no significa que la función parará cuando se llegue al final de la iteración de un array u objeto. por ejemplo:

```
function* idMaker() {
  let index = 0;
  while (index < 3)
    yield index++;
}

let gen = idMaker();
console.log(gen.next()); // { value: 0, done: false }
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: undefined, done: true }
// Accede a la variable por que se le ha dicho apesar de que el 'yield' no ha sido efectuado.
```

Es importante ver que **TypeScript** nos permitiría compilar esto y acceder a una variable que no existe. Ya que el lenguaje hasta el momento de la ejecución (tiempo de ejecución) no nos permite saber el número de veces que utilizamos el `next()`.

```
//----- main.ts -----  
function* generator(){  
    console.log('Execution started');  
    yield 0;  
    console.log('Execution resumed');  
    yield 1;  
    console.log('Execution resumed');  
}  
  
var iterator = generator();  
console.log('Starting iteration'); // Esto se ejecutará antes que nada de dentro del m  
étodo generator()  
console.log(iterator.next()); // { value: 0, done: false }  
console.log(iterator.next()); // { value: 1, done: false }  
console.log(iterator.next()); // { value: undefined, done: true }
```

```
$ node main.js  
Starting iteration  
Execution started  
{ value: 0, done: false }  
Execution resumed  
{ value: 1, done: false }  
Execution resumed  
{ value: undefined, done: true }
```

Como se puede apreciar en el ejemplo, lo primero que se ejecuta es el texto "**Starting iteration**", ya que no es hasta que ejecutemos el primer `next()`, que empezaremos a ejecutar el método `generator()`, dentro de esta función se pasa al método `yield` (si es encontrada) en cuyo caso se resume el estado del método, y se espera hasta el siguiente `next()` en el cual ejecutará el método desde el punto en el que se quedo en el estado anterior.

Este tipo de funciones también permite la inyección de variables como por ejemplo:

```
// ----- ejemplo.js -----
function* generator() {
  var bar = yield 'Console log';
  console.log(bar); // 'Un texto inyectado' -> asignado por nextThing = iterator.next('bar')
  yield 1;
}.
const iterator = generator();
// Start execution till we get first yield value
const foo = iterator.next();
console.log(foo.value); // Console log
// Resume execution injecting bar
const nextThing = iterator.next('Un texto inyectado'); // Aqui se le asigna el value a l foo.value
console.log(nextThing);
```

```
Console log
Un texto inyectado
{ value: 1, done: false }
```

Otro ejemplo de argumentos sería el siguiente:

```
function* logGenerator() {
  console.log(yield);
  console.log(yield);
  console.log(yield);
}

var gen = logGenerator();

// the first call of next executes from the start of the function
// until the first yield statement
gen.next();
gen.next('pretzel'); // pretzel
gen.next('california'); // california
gen.next('mayonnaise'); // mayonnaise
```

El siguiente ejemplo es como se tratarían las excepciones:

```
// ----- test.ts -----
function* generator() {
  try {
    yield 'foo';
    throw Error("Test");
  }
  catch(err) {
    console.log(err.message); // bar!
  }
}

var iterator = generator();
// Start execution till we get first yield value
var foo = iterator.next();
console.log(foo.value);
// como está comentado la excepción no se ejuta ya que no hay un 'next()'
//var foo = iterator.next();
```

Output

foo

Ahora veamos el output sin comentar esa línea:

```
// ----- test.ts -----
function* generator() {
  try {
    yield 'foo';
    throw Error("Test");
  }
  catch(err) {
    console.log(err.message); // bar!
  }
}

var iterator = generator();
// Start execution till we get first yield value
var foo = iterator.next();
console.log(foo.value);
var foo = iterator.next();
```

Output

foo
Test

La otra gran razón por la cual es tan importante la utilización de este tipo de métodos es porque con este tipo de métodos podemos mejorar el sistema de promesas y las peticiones asíncronas. como por ejemplo:

```
function getFirstName() {
  setTimeout(function(){
    gen.next('alex')
  }, 1000);
}

function getSecondName() {
  setTimeout(function(){
    gen.next('perry')
  }, 1000);
}

function *sayHello() {
  var a = yield getFirstName();
  var b = yield getSecondName();
  console.log(a, b); //alex perry
}

var gen = sayHello();

gen.next();
```

Esperas Asincrónicas - Async / Await

Las `async` no pueden ser utilizados en `es5` deben ser usados en `es6` o superior.

En **TypeScript** cuando código JS se está ejecutando (sincronamente) y utiliza una función la cual contiene una o más `promesas` se podrá utilizar la palabra reservada `await` para parar la ejecución del código JS hasta que la función termine correctamente, en caso de fallo, esta función generará un error de manera sincrónica que podremos atrapar mediante un `try catch`.

```
// ----- test.ts -----  
// No es código de verdad solamente es la prueba de concepto  
async function foo() {  
  try {  
    var val = await getMeAPromise();  
    console.log(val);  
  }  
  catch(err) {  
    console.log('Error: ', err.message);  
  }  
}
```

- Si la función termina entonces devolverá un valor
- Si la función falla devolverá un error que podremos capturar

Esto convierte drásticamente la programación asíncrona tan fácil como la programación sincrónica. ya que cumple 3 requisitos indispensables:

1. Capacidad de pausar la función en tiempo de ejecución
2. Capacidad de pasarle valores a funciones
3. Capacidad de lanzar excepciones en caso de fallo

El código que genera el ejemplo de arriba debería ya sonar de algo ya que esa es la sintaxis utilizada en **TypeScript** para crear `generators`, es decir, el código previamente visto se convertiría en:


```
// No olvidéis que al no tener la función que genera las peticiones
// Este código no funciona

var __awaiter = (this && this.__awaiter) || function (thisArg, _arguments, P, generator) {
  return new (P || (P = Promise))(function (resolve, reject) {
    function fulfilled(value) { try { step(generator.next(value)); } catch (e) { reject(e); } }
    function rejected(value) { try { step(generator.throw(value)); } catch (e) { reject(e); } }
    function step(result) { result.done ? resolve(result.value) : new P(function (resolve) { resolve(result.value); }).then(fulfilled, rejected); }
    step((generator = generator.apply(thisArg, _arguments)).next());
  });
};

// No es código de verdad solamente es la prueba de concepto
function foo() {
  return __awaiter(this, void 0, void 0, function* () {
    try {
      var val = yield getMeAPromise();
      console.log(val);
    }
    catch (err) {
      console.log('Error: ', err.message);
    }
  });
}
```

Obviamente esto no es lo mismo que lo que se explica en la lección anterior sino una versión más compleja.

Clases

En caso de que queramos poder instanciar una clase, necesitaremos un constructor, ya que en diferencia de lenguajes como **java** no tiene un constructor por defecto. es por eso que si queremos utilizar la palabra reservada `new` tenemos que crear un constructor con la palabra reservada `constructor` .

```
class Startup {  
  
    private text: String;  
  
    constructor (texto: String) {  
  
        this.text = texto;  
  
    }  
  
    public main () : number {  
  
        console.log(this.text);  
        return 0;  
  
    }  
  
}  
  
let s = new Startup("Hola mundo");  
s.main();
```

```
class Startup {  
  
    public static main(): number {  
  
        console.log('Hola mundo');  
        return 0;  
  
    }  
  
}  
  
Startup.main();
```

Otro ejemplo podría ser este:

Typescript

```
//----- calculos.ts -----
class Calculo {
    // variables de clase declaradas y asignadas
    private x: number = 0;
    private y: number = 0;

    // constructor
    constructor (x: number, y: number){
        // métodos del constructor
        this.setX(x);
        this.setY(y);
    }

    // Setters
    public setX(x: number) : void{
        this.x = x;
    }

    public setY(y: number) : void{
        this.y = y;
    }

    // Getters
    public getX(): number {
        return this.x;
    }

    public getY(): number {
        return this.y;
    }

    public sumar() : number {
        return (this.getX() + this.getY());
    }

    public restar() : number{
        return ( this.mayor() - this.menor() );
    }

    public menor() : number {
        if(this.getX() >= this.getY()) {
            return this.getY();
        }
        return this.getX();
    }

    public mayor() : number {
        if(this.getX() >= this.getY()) {
            return this.getX();
        }
        return this.getY();
    }
}
```

```
}  
  
let calculo = new Calculo(30,10);  
  
console.log(calculo.restar());
```

JavaScript

```
//----- calculos.js -----  
var Calculo = (function () {  
    // constructor  
    function Calculo(x, y) {  
        // variables de clase declaradas y asignadas  
        this.x = 0;  
        this.y = 0;  
        // métodos del constructor  
        this.setX(x);  
        this.setY(y);  
    }  
    // Setters  
    Calculo.prototype.setX = function (x) {  
        this.x = x;  
    };  
    Calculo.prototype.setY = function (y) {  
        this.y = y;  
    };  
    // Getters  
    Calculo.prototype.getX = function () {  
        return this.x;  
    };  
    Calculo.prototype.getY = function () {  
        return this.y;  
    };  
    Calculo.prototype.sumar = function () {  
        return (this.getX() + this.getY());  
    };  
    Calculo.prototype.restar = function () {  
        return (this.mayor() - this.menor());  
    };  
    Calculo.prototype.menor = function () {  
        if (this.getX() >= this.getY()) {  
            return this.getY();  
        }  
        return this.getX();  
    };  
    Calculo.prototype.mayor = function () {  
        if (this.getX() >= this.getY()) {  
            return this.getX();  
        }  
        return this.getY();  
    };  
    return Calculo;  
})();  
var calculo = new Calculo(30, 10);  
console.log(calculo.restar());
```

Otro ejemplo algo más visual sería el siguiente:

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
  alert(greeter.greet());
}

document.body.appendChild(button);
```

TypeScript también admite que las clases tengan propiedades estáticas compartidas por todas las instancias de la clase, la manera natural de acceder a estos atributos estáticos sería:

```
class Something {
  static instances = 0;
  constructor() {
    // Acedemos directamente mediante el nombre de la clase
    Something.instances++;
  }
}

var s1 = new Something();
var s2 = new Something();
console.log(Something.instances); // 2
```

Modificadores de clases

TypeScript también admite modificadores de accesibilidad o visibilidad como los mencionados abajo

Accesibles en	public	private	protected
Instancias de clases	Sí	No	No
clases	Sí	Sí	Sí
Clases hijas	Sí	No	Sí

Para comprender un poco mejor la tabla previamente mostrada utilizaremos el siguiente ejemplo:

```
class FooBase {
  public x: number;
  private y: number;
  protected z: number;
}

// EFECTOS EN UNA INSTANCIA
var foo = new FooBase();
foo.x; // OK
foo.y; // ERROR : private
foo.z; // ERROR : protected

// EFECTOS EN UNA CLASE HIJA
class FooChild extends FooBase {
  constructor() {
    super();
    this.x; // OK
    this.y; // ERROR: private
    this.z; // okay
  }
}
```

Abstract

`abstract` podría pensarse que también es un modificador de acceso, pero no es así, ya que no modifica la posibilidad de acceso sino que es más similar a una interfaz la cual tiene un "**contrato**" y este tipo de clases tiene una serie de normas más estrictas:

- **Una clase abstracta no puede ser directamente instanciada**, por el contrario tiene que ser utilizada mediante la herencia de dicha clase.
- La clase abstracta no puede ser accedida directamente por las clases hijas, son estas las que tienen que aportar la funcionalidad.

¿Qué es IIFE?

IIFE o Immediately-Invoked Function Expression Son funciones anónimas en las cuales estará contenido nuestro código js, esto es debido a que de esa forma estaríamos creando algo así como un namespace, es decir, el contenido de esa función no sería accesible desde fuera a no ser que se instanciará o se utilizara herencia o interfaces, el código js sería algo así:

```
// ----- point.js -----  
(function () {  
  // BODY  
  return Point;  
})();
```

Un ejemplo de esto sería el siguiente:

TypeScript

```
// Clase padre  
class Point {  
  x: number;  
  y: number;  
  constructor(x: number, y: number) {  
    this.x = x;  
    this.y = y;  
  }  
  add(point: Point) {  
    return new Point(this.x + point.x, this.y + point.y);  
  }  
}  
  
// clase hija  
class Point3D extends Point {  
  z: number;  
  constructor(x: number, y: number, z: number) {  
    super(x, y);  
    this.z = z;  
  }  
  // función add  
  add(point: Point3D) {  
    // llamada a la función del padre  
    var point2D = super.add(point);  
    // devolvemos resultado  
    return new Point3D(point2D.x, point2D.y, this.z + point.z);  
  }  
}
```

JavaScript ya compilado

```
// Método que implementa extends en TypeScript
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};

// Clase padre
var Point = (function () {
    function Point(x, y) {
        this.x = x;
        this.y = y;
    }
    Point.prototype.add = function (point) {
        return new Point(this.x + point.x, this.y + point.y);
    };
    return Point;
})();

// clase hija
var Point3D = (function (_super) {
    __extends(Point3D, _super);
    function Point3D(x, y, z) {
        _super.call(this, x, y);
        this.z = z;
    }
    // función add
    Point3D.prototype.add = function (point) {
        // llamada a la función del padre
        var point2D = _super.prototype.add.call(this, point);
        // devolvemos resultado
        return new Point3D(point2D.x, point2D.y, this.z + point.z);
    };
    return Point3D;
})(Point);
```

Se puede apreciar que la IIFE permite que **TypeScript** capture fácilmente la clase `Point` en una variable llamada `_super` y luego utilizarla en el cuerpo de la clase.

Herencia

En **TypeScript** no existe la herencia múltiple.

TypeScript

```
// Herencia

class Animal {

    name: string;

    constructor(theName: string) { this.name = theName; }

    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {

    constructor(name: string) { super(name); }

    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Rhino extends Animal {

    constructor(name: string) { super(name); }

    move(distanceInMeters = 10) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Elephant extends Animal {

    constructor(name: string) { super(name); }

    move(distanceInMeters = 20) {
        console.log("Slithering...");
    }
}
```

```
        super.move(distanceInMeters);

    }

}

// Para poder Crear un array con este typo de objetos tenemos que utilizar la clase Pa
dre Ej:

let array : Animal[] = [ new Rhino('Rinocerator'), new Snake("Serpentina"), new Elepha
nt("Elefanton") ];

// El acceso a este ejemplo sería muchísimo más sencillo
let ej2 = {Rhino: new Rhino('Rinocerator'), Snake: new Snake("Serpentina"), Elephant:
new Elephant("Elefanton")};

console.log(array);
```

JavaScript

super

En las clases hijas hace referencia a la clase padre, tenemos que tener en cuenta que `super` solo deberá ser utilizado cuando no haya de por medio **fat arrows** o funciones de flecha, ya que en ese caso debería utilizarse la palabra reservada `this` veamos unos ejemplos:

Super sin fat arrow

```
class Base {
    log() { console.log('hello world'); }
}
class Child extends Base {
    log() { super.log(); }
}
```

This con fat arrow

```
class Base {
    // realizamos una función mediante el uso de fat arrow
    log = () => { console.log('hello world'); }
}
class Child extends Base {
    logWorld() { this.log(); }
}
```

Si intentamos utilizar `super` en esta situación el compilador nos dará un error. que nos dirá que solo métodos `public` o `protected` son accesibles con la palabra reservada `super`

Sobrecarga de métodos

La sobrecarga de métodos de **TypeScript** es así:

```
function a(x: number);  
function a(x: String);  
function a(x: boolean);  
function a(x: Array<number>);  
function a(x){  
    // implementación de la función  
}
```

Este tipo de sobrecarga no tiene mucho sentido porque sería más simple poner un **Any**.

Mixin

Cuando tenemos una serie de clases de **TypeScript** las cuales no tienen ninguna relación de herencia entre ellas pero necesitamos que tengan comportamientos comunes, crearemos **mixin**, para los que hayan visto **PHP** es algo así como los **traits**, estos **mixin** nos permiten utilizar las interfaces como clases para poder intentar tener un sucedaneo de herencia multiple, la cual nos permitirá modificar el comportamiento de las clases para poder implementar estos "**contratos**".

Esto es un mixin en **TypeScript** a continuación vamos a explicar paso a paso el siguiente ejemplo:

```
// Disposable Mixin
class Disposable {
  isDisposed: boolean;
  dispose() {
    this.isDisposed = true;
  }
}

// Activatable Mixin
class Activatable {
  isActive: boolean;
  activate() {
    this.isActive = true;
  }
  deactivate() {
    this.isActive = false;
  }
}

class SmartObject implements Disposable, Activatable {
  // Este constructos lo que hará será que mostrará por pantalla
  // los estados isActive y isDisposed cada 500 ms
  constructor() {
    setInterval(() => console.log(this.isActive + " : " + this.isDisposed), 500);
  }

  interact() {
    this.activate();
  }

  // Disposable
  isDisposed: boolean = false;
  dispose: () => void;
  // Activatable
  isActive: boolean = false;
  activate: () => void;
```

```

    deactivate: () => void;
  }
  applyMixins(SmartObject, [Disposable, Activatable]);

  let smartObj = new SmartObject();
  // esto generará una interacción cada segundo cambiando el estado de
  // `false - false` a `true - true` la primera vez,
  // luego como la variable ya valdría `true - true`
  // simplemente se dedicaría a mostrarla cada segundo
  setTimeout(() => smartObj.interact(), 1000);

  ////////////////////////////////////////////////////
  // In your runtime library somewhere
  ////////////////////////////////////////////////////

  function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
      Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
        derivedCtor.prototype[name] = baseCtor.prototype[name];
      });
    });
  }
}

```

En este trozo de código podremos apreciar que hay dos clases distintas y que cada una de ellas se enfoca en una actividad o capacidad distinta. Más adelante mezclaremos estas clases juntas en otra clase para poder obtener ambas capacidades.

```

// Disposable Mixin
class Disposable {
  isDisposed: boolean;
  dispose() {
    this.isDisposed = true;
  }
}

// Activatable Mixin
class Activatable {
  isActive: boolean;
  activate() {
    this.isActive = true;
  }
  deactivate() {
    this.isActive = false;
  }
}

```

Lo siguiente que haremos es crear una clase la cual implementará la combinación entre las dos clases previamente vistas.


```
class SmartObject implements Disposable, Activatable {
```

Lo primero que podreis apreciar es que en vez de utilizar `extends` estamos utilizando `implements`, esto es debido a que en **TypeScript** no existe la herencia multiple y utilizando la palabra reservada `implements` trata a las clases que se está implementando como interfaces que deben ser cumplidas. Esto implica que la clase que acabamos de crear debe tener los mismo métodos que las dos "interfaces"

```
// Implementamos Disposable
isDisposed: boolean = false;
dispose: () => void; // esto es lo mismo que declarar una función vacía

// Implementamos Activatable
isActive: boolean = false;
activate: () => void;
deactivate: () => void;
```

El siguiente paso es crear una función la cual nos permitirá realizar el mixin de cada una de las propiedades de las clases en el objeto que deseemos

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
      derivedCtor.prototype[name] = baseCtor.prototype[name];
    });
  });
}
```

En el siguiente paso lo que se hace es realmente implementar el mixin de las clases:

```
applyMixins(SmartObject, [Disposable, Activatable]);
```

Interfaces

En TypeScript las interfaces son las encargadas de comprobar los tipos de las variables que se pasan como argumentos y del cumplimiento de los contratos.

```
interface a {  
    b: number;  
}  
  
interface b extends a {  
    c: string;  
}  
  
class test implements b {  
    b: number;  
    c: string;  
    constructor (b: number, c: string) {  
        this.b = b;  
        this.c = c;  
    }  
}
```

Decorators

Para poder utilizar los decoradores tenemos que permitirlo en nuestro `tsconfig` y escribir lo siguiente

```
"experimentalDecorators": true
```

La sintaxis de los decoradores es

```
@miDecorador
```

```

function log(constructor: Function): void{
    console.log('Registered Class: ' + constructor['name'] + ' at ' + Date.now());
}

function logm<T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>): void {
    console.log('Registered Method: ' + propertyKey.toString() + ' at ' + Date.now());
}

function logparam (target: Object, propertyKey: string | symbol, parameterIndex: number): void {
    console.log('Registered Parameter: ' + propertyKey.toString() + ' - ' + parameterIndex + ' at ' + Date.now());
}
// Así se utiliza un decorador
@log
class MyClass {

    public name: string = 'name';

    constructor() {
        console.log('constructor');
    }

    @logm
    public myMethod() {
        console.log('method')
    }

    @logm
    public myMethod2(param1: number, @logparam param2: boolean) {
        console.log('method2')
    }
}

var myClass = new MyClass();
myClass.myMethod();
myClass.myMethod2(1, false);

// ----- COMO SE USAN LOS DECORADORES
// type ClassDecorator = <TFunction extends Function>(target: TFunction): TFunction | void;
// type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>): TypedPropertyDescriptor<T> | void;
// type PropertyDecorator = (target: Object, propertyKey: string | symbol): void;
// type ParameterDecorator = (target: Object, propertyKey: string | symbol, parameterIndex: number): void;

```


Class Decorator

```
function ClassDecoratorParams(param1: number, param2: string) {  
    return function(  
        target: Function // The class the decorator is declared on  
    ) {  
        console.log("ClassDecoratorParams(" + param1 + ", '" + param2 + "') called on:  
", target);  
    }  
}  
  
@ClassDecoratorParams(1, "a")  
@ClassDecoratorParams(2, "b")  
class ClassDecoratorParamsExample {  
}
```

```
ClassDecoratorParams(2, 'b') called on: function ClassDecoratorParamsExample() {  
}  
ClassDecoratorParams(1, 'a') called on: function ClassDecoratorParamsExample() {  
}
```

Property Decorator

```
function PropertyDecorator(  
  target: Object, // The prototype of the class  
  propertyKey: string | symbol // The name of the property  
) {  
  console.log("PropertyDecorator called on: ", target, propertyKey);  
}  
  
class PropertyDecoratorExample {  
  @PropertyDecorator  
  name: string;  
}
```

```
PropertyDecorator called on: {} name
```

Method Decorator

```
function MethodDecorator(  
    target: Object, // The prototype of the class  
    propertyKey: string, // The name of the method  
    descriptor: TypedPropertyDescriptor<any>  
    ) {  
    console.log("MethodDecorator called on: ", target, propertyKey, descriptor);  
}  
  
class MethodDecoratorExample {  
    @MethodDecorator  
    method() {  
    }  
}
```

```
MethodDecorator called on: { method: [Function] } method { value: [Function],  
    writable: true,  
    enumerable: true,  
    configurable: true  
}
```


Static Method Decorator

```
function StaticMethodDecorator(  
  target: Function, // the function itself and not the prototype  
  propertyKey: string | symbol, // The name of the static method  
  descriptor: TypedPropertyDescriptor<any>  
) {  
  console.log("StaticMethodDecorator called on: ", target, propertyKey, descriptor);  
}  
  
class StaticMethodDecoratorExample {  
  @StaticMethodDecorator  
  static staticMethod() {  
  }  
}
```

```
StaticMethodDecorator called on: function StaticMethodDecoratorExample() {  
}
```

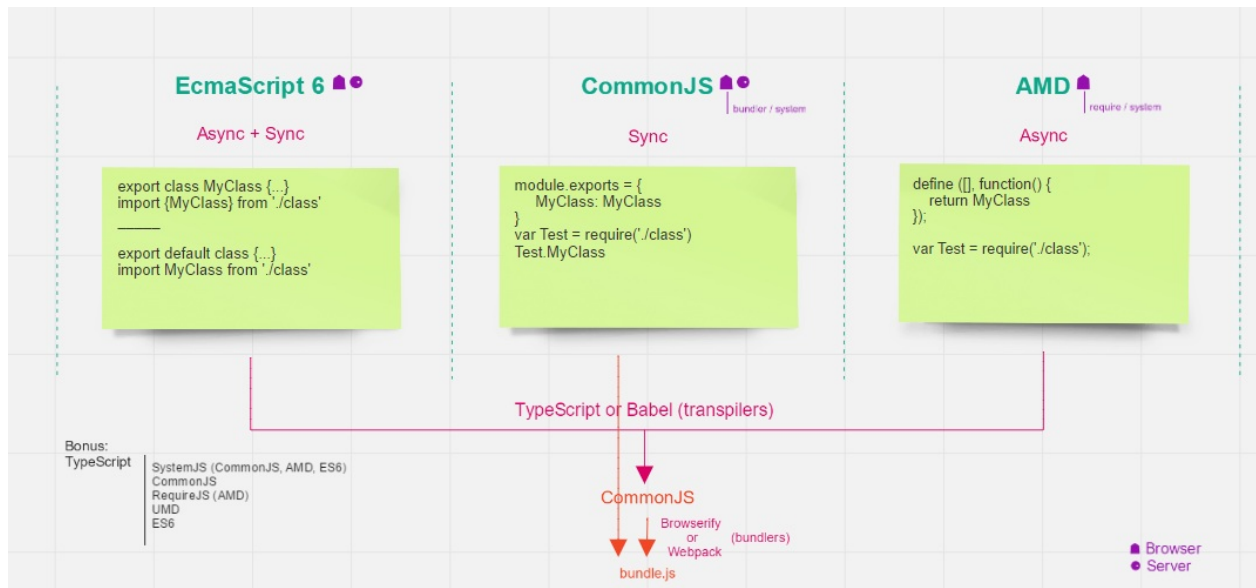
Parameter Decorator

```
function ParameterDecorator(  
  target: Function, // The prototype of the class  
  propertyKey: string | symbol, // The name of the method  
  parameterIndex: number // The index of parameter in the list of the function's parameters  
) {  
  console.log("ParameterDecorator called on: ", target, propertyKey, parameterIndex)  
;  
}  
  
class ParameterDecoratorExample {  
  method(@ParameterDecorator param1: string, @ParameterDecorator param2: number) {  
  }  
}
```

```
ParameterDecorator called on: { method: [Function] } method 1  
ParameterDecorator called on: { method: [Function] } method 0
```

Tipos de carga de módulos

En la actualidad existen tres tipos de cargas de módulos, los módulos son una manera de llamar a las clases o métodos que deseamos exportar para que otras clases puedan utilizarlas mediante importaciones.



EcmaScript 6

Es un lenguaje del lado del cliente y del servidor, es **Async + Sync**. Según las especificaciones de ES6 existen 2 tipos de exportación:

- exportación con nombre (varias por módulo)
- exportación por defecto (una por módulo). de módulos, unos ejemplos serían:

Exportación con nombre (varias por módulo) - Permite la carga de varias clases separadas por comas, a la hora de especificar en la variable a que clase se hace referencia no hace falta poner el **.js**

Ejemplo 1:

```
export class MyClass {...}
import {MyClass} from './class'
```

Ejemplo 2 (Javascript):

```
//----- calculos.js -----  
export function sumar(x, y) {  
  return x + y;  
}  
export function restar(x, y) {  
  return x - y;  
}
```

```
//----- main.js -----  
import { sumar, restar } from 'calculos';  
console.log(sumar(2, 3)); // 5  
console.log(restar(4, 3)); // 1
```

Este tipo de carga también permite que se puedan exportar todos los módulos enteros y acceder a las exportaciones con nombre usando la siguiente sintaxis:

```
//----- main.js -----  
import * as calc from 'calculos';  
console.log(calc.sumar(2, 3)); // 7  
console.log(calc.restar(4, 3)); // 1
```

Exportación por defecto (una por módulo) - Los módulos que solo exporten un único valor son muy populares en la comunidad de Node.js y en el desarrollo frontend. Un módulo de ES6 puede tomar un valor por defecto en la exportación (**más información de como es generado el default abajo**).

Ejemplo 1:

```
export default class {...}  
import MyClass from './class'
```

Ejemplo 2 (Javascript):

```
//----- miFunc.js -----  
export default function () { ... };
```

```
//----- main1.js -----  
import miFunc from 'miFunc';  
miFunc();
```

En ES6 también podemos hacer esto usando export seguido de una definición estándar de lo que vamos a exportar.

```
//----- modulo.js ----- (PARA IMPORTAR)
export const SECRETO = "Me gustan los robots";

export var opciones = {
  color: 'rojo',
  imagen: false,
};

export function suma(a, b) {
  return a + b;
}
```

Lo interesante es que el nombre que le des será el mismo usara para importar esa parte del módulo.

En caso de que este objeto fuera importado en otro archivo el resultado sería el siguiente:

```
//----- importado.js -----
const SECRETO = "Me gustan los robots";

var opciones = {
  color: 'rojo',
  imagen: false,
};

function suma(a, b) {
  return a + b;
}

export default {
  SECRETO,
  opciones,
  suma,
};
```

Es importante apreciar que cuando realizamos la importación se crea un export default cuando vamos a importar una clase

```
export default {
  SECRETO,
  opciones,
  suma,
};
```

A primera vista, tener módulos de forma nativa en ES6 puede parecer una funcionalidad inútil, después de todo ya existe varios sistemas de módulos muy buenos. Pero los módulos de ES6 tienen características que no se pueden agregar con una librería, como una sintaxis

reducida y una estructura estática de módulos (lo que ayuda a optimizar entre otras cosas). Además se espera que termine con la fragmentación entre los sistemas CommonsJS y AMD.

Tener un único estándar para módulos nativo significa:

- No más UMD (Definición Universal de Módulos): UMD es un patrón que permite que un mismo archivo sea usado por distintos sistemas (como CommonJS y AMD). Cuando ES6 este listo para su uso UMD se volverá obsoleto.
- Las nuevas API de los navegadores se volveran módulos en lugar de variables globales o propiedades como navigator.
- No más objetos como namespaces. Objetos como Math y JSON sirven como namespaces para funciones en ECMAScript 5. En el futuro, esta funcionalidad la darán los módulos.

[Import ES6](#) [Exports ES6](#)

CommonJS

Es una librería que esta hecha principalmente para el lado del servidor y el cliente, es **Sync**. Un ejemplo del código JS es:

Ejemplo 1 (javascript):

```
//----- exportación.js -----  
module exports = {  
  MyClass: MyClass  
}
```

```
//----- main.js -----  
var Test = required('./exportación')  
Test.MyClass
```

Ejemplo 2 (javascript):

```
//----- exportación.js -----  
var multiplicar = function(x){  
    return x * 3;  
}  
  
var sumar = function (x) {  
    return x + 2;  
}  
  
exports.multiplicar = multiplicar;  
exports.sumar = sumar;  
  
// Código añadido...  
var saludo = 'Hola mundo';  
module.exports = saludo;
```

```
//----- main.js -----  
var importacion = require('./importacion');  
  
console.log('Este es el resultado de la importación ' + importacion);  
  
var tipo = typeof importacion;  
  
console.log('Este es el tipo del contenido importado ' + tipo);
```

El resultado de la ejecución de main.js en un servidor nodejs sería algo así:

```
Este es el resultado de la importación Hola mundo  
Este es el tipo del contenido importado string  
  
/Users/usuario/Projects/Importacion/main.js:12  
var multiplicacion = importacion.multiplicar(5);  
                                ^  
TypeError: undefined is not a function
```

Ejemplo 3: (javascript)

```
//----- HealthComponent.js -----  
var HealthComponent = function (initialHealth) {  
  
    this.health = initialHealth;  
  
    this.getHit = function (amount) {  
  
        this.health -= amount;  
  
    }  
  
    return this;  
}  
  
module.exports = HealthComponent;
```

```
//----- main.js -----  
var HealthComponent = require('./HealthComponent.js');  
  
var myHealthComponent = new HealthComponent(10);
```

También se podría importar la clase primero y luego la instanciarla con un valor de vida inicial. Ejemplo:

```
var myHealthComponent = require('./HealthComponent.js')(10);
```

AMD - RequirerJS / SystemJS

Es una librería solo del lado del cliente (navegador), es **Async**. Un ejemplo del código JS es:

```
define ([], function(){  
    return MyClass  
});  
  
var Test = require('./class');
```

Para utilizar ECmaScript6 o AMD se suelen utilizar transpiladores en nuestro caso emplearemos **TypeScript** pero también se podrían utilizar otros como por ejemplo Babel.

Funcionamientos de los módulos

1. Primero, crearíamos nuestros módulos los exportaríamos y los incluiríamos en los archivos que quieres incluirlos
2. El segundo paso sería utilizar transpiladores como por ejemplo **TypeScript o Babel** para convertir esas exportaciones e importaciones en javascript.
3. El último paso sería la utilización de **bundlers**, estos **bundlers** intentan imitar

¿Qué son bundlers?

En la actualidad no todos los navegadores soportan estos tipos de carga de módulos es por eso que se suele utilizar **bundlers** para concatenar, unificar y minimizar los **.js** finales.

Los dos más utilizados son:

- [webpack](#)
- [browserify](#)

Descargado en: www.detodoprogramacion.org

webpack

Su utilidad reside en la fragmentación de código: no todas las partes de una webapp requieren todo el código JavaScript, por eso se encarga de cargar sólo las partes necesarias en cada petición. Además, funciona con un gran número de lenguajes de plantilla y preprocesadores de JavaScript (TypeScript o CoffeeScript, ReactJS...) y CSS (LESS, SASS...). Para que importar diferentes componentes sea sencillo e intuitivo, **Webpack** implementa el ya estandarizado RequireJS para la inyección de dependencias de nuestra aplicación. Ejemplo:

Primero empezaremos con la instalación de **webpack**:

```
npm install -g webpack
```

HTML5

```
<!DOCTYPE html>
<html lang="ES">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Introducción a Webpack</title>
    <link rel="stylesheet" href="">
  </head>
  <body>

    <!-- carga del script -->
    <script type="text/javascript" src="script.js" charset="utf-8"></script>
  </body>
</html>
```

Javascript

```
//----- mensajes.js -----
var mensaje = "Hola, qué tal?"
exports.saludo = mensaje
```

```
//----- entry.js -----
var mensajes = require('./mensajes.js')
document.write(mensajes.saludo)
```

Ahora, desde la línea de comandos, ejecutamos el compilado.

```
$ webpack ./entry.js script.js
```

Si este ejemplo no ha sido suficientemente explicativo vea el siguiente enlace [webpack-howto](#)

browserify

Browserify es una librería de Node.js, escrita por substack, uno de los mayores contribuidores al core de Node.js y con mayor número de módulos publicados en NPM.

Nos permite escribir código JavaScript del cliente, como si estuviésemos programando en Node.js, es decir, como por ahora no tenemos módulos de forma nativa en JavaScript (hasta que se implante el estándar ECMAScript6 que si los trae) hay librerías que se encargan de imitar ese comportamiento (Caso de Require.js hace un tiempo).

Con esta librería, podemos instalar módulos para el cliente, con NPM, al igual que hacíamos con bower, pero ahora, podemos llamarlos desde nuestro código con la palabra reservada **require**, al igual que en Node.js

Ejemplo: Vamos a crear **app.persona.js** como módulo para ver como se exportaría e importaría en nuestra aplicación con browserify.

```
// source/scripts/app.persona.js
var Persona = function(nombre, edad) {
  this.nombre = nombre;
  this.edad = edad;

  var self = this;

  return {
    saludar: function() {
      alert("Hola, mi nombre es " + self.nombre);
    },
    presentar: function() {
      alert("Tengo " + self.edad + " años.");
    }
  };
};

module.exports = Persona;
```

Exportamos la variable Persona como módulo, y que devuelve 2 funciones, saludar y presentar, que son las únicas que podremos utilizar cuando las llamemos desde app.main.js

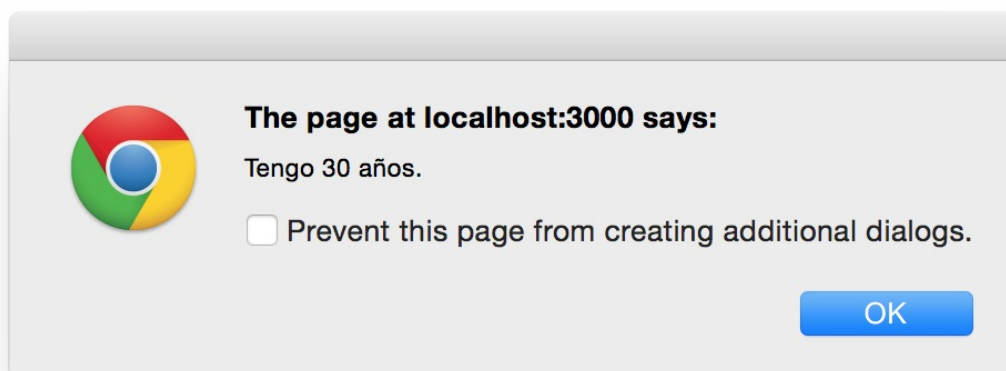
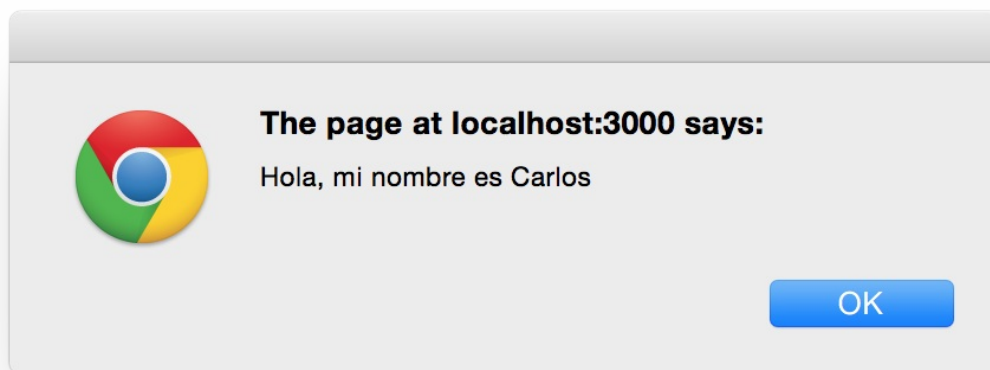
```
var $ = require('jquery');
var persona = require('./app.persona');

$('h1').html('Hola Browserify');

var carlos = new persona("Carlos", 30);
carlos.saludar();
carlos.presentar();
```

Importamos el módulo recién creado con require('./app.persona') indicándole la ruta donde está el archivo.

Creamos un objeto persona en la variable carlos, le pasamos los parámetros Carlos y 30 como nombre y edad. y llamamos a las funciones saludar() y presentar() que mostrarán una alerta JavaScript como las siguientes:



Como funciona TypeScript con módulos

La sintaxis de **TypeScript** A la hora de relizar un módulo se parece mucho a la de **ES6** pero se puede especificar que tipo de módulo deseamos obtener en el javascript compilado, es decir, que especificación deseamos obtener al final. Por defecto se utiliza commonjs.

Tipo de módulos

Módulo global

Por defecto cuando escribes un archivo TypeScript como por ejemplo un archivo llamado `a.ts` :

```
//----- a.ts -----  
var a = 123;
```

Si creáramos otro archivo en el mismo proyecto por ejemplo `b.ts` y quisiéramos utilizar variables del archivo `a.ts` podríamos hacerlo ya que para TypeScript ambos pertenecen al namespace global.

```
//----- b.ts -----  
var b = a; // Permitido
```

No es necesario decir que utilizar un namespace global es peligroso ya que esto podría dar lugar a muchos conflictos de nombre.

Módulos de archivo

También conocidos como **módulos externos**. Si utilizamos las palabras reservadas `import` o `export` dentro de un fichero TypeScript estás creando un ámbito local de este archivo. Esto lo que hace es preveer el namespace global. Si volvemos al ejemplo anterior:

```
//----- a.ts -----  
export var a = 123;
```

```
//----- b.ts -----  
var b = a; // NO permitido - no encontraría el valor de a
```

Si quisieras obtener el valor de la variable `a` necesitarías utilizar la siguiente sintaxis:

```
//----- b.ts -----  
import {a} from "./a";  
var b = a; // Permitido, ahora sí encontraría el valor
```

Utilizar `import` para variables exportadas solo sirve para quitarlo del namespace global.

La manera de generar en TypeScript módulos externos es utilizando la palabra reservada `module`, mediante la cual podremos encapsular una parte de nuestro código dentro de un módulo.

```
//----- a.ts -----  
// Se puede realizar un módulo sin emplear declare  
declare module "a" {  
    // Contenido de la clase  
  
    export var a:number; /*sample*/  
}  
declare var hello = 'Hello world';
```

```
//----- b.ts -----  
import var a = require("./a");  
var b = a;
```

Puede haber varios módulos con el mismo nombre en distintos namespace y si quisieramos incluirlos los dos podríamos renombrarlos.

Es importante observar la palabra reservada `declare` la cual nos permitira a la hora de importar este archivo tener todas las variables delcaradas accesibles, de este modo podríamos acceder a la variable hello como si estuviera en el archivo `b.ts`.

Namespace

Contexto aislado del resto con lo que podemos trabajar, la diferencia entre `module` y **namespace** es que un módulo estára normalmente dentro de un archivo y un namespaces puede ser un conjunto de archivos, permitiendonos así englobar una serie de clases(*archivos*) bajo un mismo namespace. **En la actualidad el namespace es considerado un módulo interno.**

Exportación

```
//----- namespace.ts -----  
// Módulos internos TypeScript  
namespace MySpace {  
    export class MyClass {  
        public static myProperty: number = 1;  
    }  
}
```

Existen dos sintaxis para la importación de namespace:

Importación

```
//----- importacionReference.ts -----  
/// <reference path="namespace.ts" />  
console.log(MySpace.MyClass.MyProperty);
```

```
//----- importacionImport.ts -----  
import {MySpace} from 'namespace.ts'; // Cuidado  
console.log(MySpace.MyClass.MyProperty);
```

La importación en los namespaces tiene como sintaxis recomendada la siguiente: `///`

`<reference ... />` ,

Definitions

Son archivos **.d.ts**, los cuales contendrán las **"reglas"** que deberemos cumplir para poder utilizar adecuadamente librerías externas de javascript, como por ejemplo jQuery. Los **.d.ts** suelen ser interfaces.

Los archivos **d.ts**, no se generan de forma automática, pero existe un repositorio en github con todos los distintos **d.ts** el repositorio se llama **typings** estos funcionan de la siguiente forma:

```
# Install Typings CLI utility.
npm install typings --global

# Search for definitions.
typings search tape

# Find a definition by name.
typings search --name react

# If you use the package as a module:
# Install non-global typings (defaults to "npm" source, configurable through `defaults
source` in `.typingsrc`).
typings install debug --save

# If you use the package through `
```


Sistemas de automatización

Porque hoy en día el flujo de trabajo de un desarrollador se ha vuelto más complejo, usamos muchas herramientas de desarrollo, por lo cual configurar cada tarea y ejecutarla “manualmente” y por separado requiere demasiado tiempo, para solucionar este problema tenemos sistemas de automatización de tareas.

Gulp.js

Gulp.js es un build system(sistema de construcción) que permite automatizar tareas comunes de desarrollo, tales como la minificación de código JavaScript, recarga del navegador, compresión de imágenes, validación de sintaxis de código y un sin fin de tareas más.

Descargado en: www.detodoprogramacion.org

¿Cómo funciona Gulp.js?

Gulp.js utiliza el módulo Stream de Node.js, lo cual lo hace más rápido para construir, a diferencia de **Grunt.js**.

Gulp.js no necesita escribir archivos y/o carpetas temporales en el disco duro, lo cual supone que realizará las mismas tareas que **Grunt.js** pero en menor tiempo.

Gulp.js utiliza el método `pipe()`, este método obtiene todos los datos de un Stream legible(readable) y lo escribe en destino que le indiquemos ó de lo contrario lo entrega o sirve hacia otro pipe.

En la siguiente imagen veremos como **Grunt.js** manipula los archivos al realizar sus tareas:



Y en esta veremos como **Gulp.js** manipula los archivos al realizar sus tareas:



Como podemos ver, aunque los 2 hicieron la misma tarea Gulp.js no escribió archivos temporales en el disco duro. **Gulp.js realizó 4 operaciones y en cambio Grunt.js realizó 8 operaciones.**

Gulp.js utiliza el poder del paquete Node.js **vinyl-fs** para leer y escribir Streams.

Gulp.js también utiliza un paquete Node.js para la secuenciación, ejecución de tareas y dependencias en máxima concurrencia, llamado **Orchestrator**.

Más adelante veremos con mayor detalle como trabajan los Streams de Node.js.

¿Cómo instalar Gulp.js?

Para instalarl **gulp.js** de modo global

```
npm install -g gulp
```

Si estás usando Linux o Mac, tal vez necesites anteponer la palabra sudo para poder ejecutar este comando con los permisos de administrador, así:

```
sudo npm install -g gulp
```

Verificamos que Gulp.js ha sido instalado correctamente.

```
gulp -v
```

Si lo tenemos instalado correctamente, nos mostrará lo siguiente:

```
CLI version 3.9.1
```

¿Cómo usar Gulp.js?

Una vez instalado en nuestro sistema estamos listos para crear nuestro primer proyecto usando Gulp.js, nuestro pequeño proyecto concatenará dos archivos .js en uno solo y luego lo minificará. Así que configuraremos 2 tareas(concatenar y minificar), todo esto contenido en una tarea llamada “demo”.

Creamos una carpeta llamada: **gulp-primeros-pasos**, ingresamos a esa carpeta mediante terminal.

Luego allí dentro creamos nuestro archivo: **gulpfile.js**, que es el archivo que Gulp.js necesita para saber que tareas realizará y de momento no le podremos ningún contenido.

Luego escribimos lo siguiente(en este punto suponemos que tenemos instalado Node.js)

```
npm init
```

Npm nos pedirá los datos de nuestro proyecto, ya que en esta ocasión sólo estamos haciendo un demo. Simplemente presionaremos Enter a todas las preguntas.

Con esto, Npm nos debe haber creado un archivo llamado: package.json, que contendrá algo parecido a lo siguiente:

```
{
  "name": "gulp-primeros-pasos",
  "version": "0.0.1",
  "description": "Gulp: Primeros pasos",
  "main": "gulpfile.js",
  "author": "jansanchez",
  "license": "MIT"
}
```

Ahora agregaremos las dependencias de desarrollo a nuestro proyecto, la primera a instalar será: gulp, así que escribimos lo siguiente en nuestra terminal:

```
npm install --save-dev gulp
```

Luego instalamos: [gulp-concat](#)

```
npm install --save-dev gulp-uglify
```

Tengamos en cuenta que sí no agregamos el parámetro: `--save-dev`, entonces Npm no agregará este paquete como una dependencia de desarrollo de nuestro proyecto y mucho menos lo agregará a nuestro archivo `package.json`.

Como podremos observar, nuestro archivo `package.json` a cambiado y debería contener algo parecido a lo siguiente:

```
{
  "name": "gulp-primeros-pasos",
  "version": "0.0.1",
  "description": "Gulp: Primeros pasos",
  "main": "gulpfile.js",
  "author": "jansanchez",
  "license": "MIT",
  "devDependencies": {
    "gulp": "^3.8.7",
    "gulp-concat": "^2.3.4",
    "gulp-uglify": "^0.3.1"
  }
}
```

Como vemos, se agregó la clave **devDependencies** y en su interior se comienzan a guardar nuestras dependencias de desarrollo y la versión que hemos instalado localmente.

Luego vamos a crear las siguientes carpetas y archivos:

Creamos la carpeta **js** y dentro de esta carpeta crearemos la carpeta **source**.

Dentro de la carpeta **source** crearemos el archivo **1.ts** y le agregaremos el siguiente contenido:

```
// contenido del archivo 1.js
var sumar = function (a, b){
  return a + b;
};
```

Nuevamente dentro de la carpeta **source** crearemos el archivo **2.js** y le agregaremos el siguiente contenido:

```
// contenido del archivo 2.js
var restar = function (a, b){
  return a - b;
};
```

Ahora vamos a poner el siguiente contenido a nuestro archivo **gulpfile.js**.

```
/*
 * Dependencias
 */
var gulp = require('gulp'),
    concat = require('gulp-concat'),
    uglify = require('gulp-uglify');

/*
 * Configuración de la tarea 'demo'
 */
gulp.task('demo', function () {
  gulp.src('js/source/*.js')
    .pipe(concat('todo.js'))
    .pipe(uglify())
    .pipe(gulp.dest('js/build/'))
});
```

Con esto ya tenemos todo configurado, así que para ponerlo a prueba en nuestra terminal escribimos lo siguiente:

```
gulp demo
```

Y si todo anda bien, nos dará el siguiente mensaje:

```
[11:23:09] Using gulpfile ~/htdocs/gulp-primeros-pasos/gulpfile.js
[11:23:09] Starting 'demo'...
[11:23:09] Finished 'demo' after 9 ms
```

El cual nos indica que la tarea demo se ejecutó con éxito en 9 milisegundos.

Para comprobar si se ejecutaron las 2 tareas requeridas, nos dirigimos a la carpeta **js/build** y abrimos el archivo **todo.js** y nos debe mostrar el siguiente contenido:

```
var sumar=function(r,n){return r+n},restar=function(r,n){return r-n};
```

Como vemos, con unas simples y limpias lineas de código hemos realizado 2 tareas de desarrollo comunes(concatenar y minificar archivos .js).

Analizando el gulpfile.js

Ahora vamos a analizar el código que escribimos en nuestro **gulpfile.js** para entender un poco más como funciona Gulp.js.

Primero para llevar a cabo las tareas que deseamos, requerimos los siguientes paquetes: gulp, gulp-concat y gulp-uglify, así:

```
/*
 * Dependencias
 */
var gulp = require('gulp'),
    concat = require('gulp-concat'),
    uglify = require('gulp-uglify');
```

API - Documentación

Gulp.js tiene una pequeña API, esto te permitirá aprender Gulp.js rápidamente.

GULP.TASK()

Con el método **gulp.task()** definimos una tarea, este método toma 3 argumentos: *el nombre de la tarea, la ó las tareas de las que depende esta tarea y la función que ejecutará al llamar esta tarea.*

En nuestro ejemplo sólo usamos 2 parámetros: el nombre y la función, así:

```
/*
 * Configuración de la tarea 'demo'
 */
gulp.task('demo', function () {
  // Contenido de la tarea 'demo'
});
```

Con lo cual declaramos “demo” como nombre de la tarea y dentro escribimos lo que deseamos que haga nuestra tarea.

Así que si queremos llamar esta tarea tan solo escribimos en nuestra terminal:

```
gulp demo
```

Lista de tareas

Una tarea también puede actuar como una lista de tareas, supongamos que queremos definir una tarea que corra otras 3 tareas por ejemplo: imágenes, css y js. Entonces escribiríamos lo siguiente:

```
gulp.task('estaticos', ['imagenes', 'css', 'js']);
```

Lo que quiere decir que al ejecutar la tarea “estaticos” con el comando **gulp staticos** se ejecutarán estas 3 tareas.

El detalle es que estas tareas corran asincrónicamente, osea que correrán todas juntas al mismo tiempo sin ningún orden de ejecución.

Tareas como dependencia

Si deseamos que una tarea se ejecute **sí y solo sí** otra tarea haya terminado antes, entonces podemos hacer lo siguiente:

```
gulp.task('css', ['imagenes'], function () {  
  /*  
   * Aquí iría el contenido de mi tarea 'css'  
   * Que se ejecutará solo cuando la tarea  
   * 'imagenes' haya terminado.  
   */  
});
```

Entonces, cuando corramos la tarea “css”, Gulp.js ejecutará primero la tarea “imagenes”, esperará a que esta tarea termine y luego recién ejecutará la tarea “css”.

TAREA POR DEFECTO(DEFAULT)

Gulp.js nos permite definir una tarea por defecto, que corra tan solo al escribir el comando **gulp**. Esto se puede hacer tan solo poniendole a la tarea el nombre de **default**, así:

```
gulp.task('default', function () {  
  /*  
   * Código de nuestra tarea por defecto.  
   */  
});
```

Y claro, también puedes hacer que tu tarea por defecto sea una lista de tareas, así:

```
gulp.task('default', ['css', 'js']);
```

Esta tarea ejecutará las tarea ‘css’ y ‘js’, tan solo escribiendo en nuestra terminal:

```
gulp
```

gulp.src()

El método **gulp.src()** toma como parámetro un valor glob es decir, una cadena que coincida con uno o más archivos usando los patrones que usa el intérprete de comandos de unix(shell) y retorna un stream que puede ser “pipeado” a un plugin adicional ó hacia el método **gulp.dest()**.

Este parámetro puede ser una cadena o una colección(Array) de valores glob.

Gulp.js usa el paquete de Node.js node-glob para obtener los archivos especificados en él ó los globs ingresados.

Ejemplos de globs

- `js/source/1.js` coincide exactamente con el archivo.
- `js/source/*.js` coincide con los archivos que terminen en .js dentro de la carpeta js/source.
- `js/**/*.js` coincide con los archivos que terminen en .js dentro de la carpeta js y dentro de todas sus sub-carpetas.
- `!js/source/3.js` Excluye específicamente el archivo 3.js.
- `static/*.(js|css)` coincide con los archivos que terminen en .js ó .css dentro de la carpeta static. Existen más patrones, los puedes revisar desde la documentación de la librería minimatch.

Así que tienes la oportunidad de realizar todas las combinaciones posibles, según lo necesites.

Como en nuestra demo, necesitábamos encontrar todos los archivos que terminen en .js dentro de la carpeta **js/source**, así:

```
gulp.src('js/source/*.js')
```

Cada vez que Gulp.js encuentre un archivo que coincida con nuestro patrón, lo irá metiendo dentro de un Stream, que será como una colección de archivos. Claro, respetando las propiedades de cada archivo(ruta, etc).

Entonces podemos decir que tendremos todos esos archivos con sus respectivas propiedades dentro de un Stream, Este Stream puede ser manipulado por Gulp.js.

El método pipe() de Node.js

El método pipe() puede leer, ayudar a transformar y grabar los datos de un Stream.

Es por eso que en nuestro ejemplo usamos el método pipe() 3 veces.

La primera vez lo usamos para leer el Stream y se lo pasamos al plugin “concat” para que este realice la concatenación y así transforme los datos del Stream, así:

```
.pipe(concat('todo.js'))
```

La segunda vez lo usamos para leer los datos actuales(js concatenado) y se lo pasamos al plugin “uglify”, para que realice la minificación del archivo concatenado. Todo esto sin escribir en el disco ningún archivo temporal, así:

```
.pipe(uglify())
```

La tercera vez se lo pasamos a el método **gulp.dest()**, así que veamos que hace este método.

gulp.dest()

Canaliza y escribe archivos desde un Stream, por lo que puede canalizar a varias carpetas. Creará las carpetas que no existan y retornará el Stream, por si deseamos realizar alguna acción más.

En pocas palabras, sirve para escribir los datos actuales de un Stream.

Y en nuestro ejemplo lo usamos así:

```
.pipe(gulp.dest('js/build/'))
```

Con lo cual escribimos los datos resultantes del Stream dentro de la carpeta **js/build/**.

El código final nos quedó así:

```
/*
 * Dependencias
 */
var gulp = require('gulp'),
    concat = require('gulp-concat'),
    uglify = require('gulp-uglify');

/*
 * Configuración de la tarea 'demo'
 */
gulp.task('demo', function () {
  gulp.src('js/source/*.js')
    .pipe(concat('todo.js'))
    .pipe(uglify())
    .pipe(gulp.dest('js/build/'))
});
```

Así como realicé 2 tareas consecutivas, con Gulp.js se pueden realizar muchas más.

gulp.watch()

Ver archivos y hacer algo cuando se modifique un archivo. Esto siempre devuelve un EventEmitter que emite los eventos de cambio.

Tiene 2 formas de usar:

gulp.watch(glob, tareas) ó gulp.watch(glob, callback).

```
gulp.watch('js/source/*.js', ['js']);
```

Con lo cual, cada vez que se modifique un archivo .js que se encuentre dentro de la carpeta **js/source/** automáticamente se ejecutará la tarea **js**.

```
gulp.watch('js/source/*.js', function(){
  /*
   * Aquí iría el código de la acción que deseas realizar,
   * Cuando hayan cambios en dichos archivos.
   *
   * También podrías ejecutar una tarea mediante el método
   * gulp.start('js')
   *
   * Pero este método no es oficial, le pertenece al
   * paquete 'Orchestrator' ya que Gulp hereda los
   * métodos de 'Orchestrator'.
   */
});
```


Consejos

En este apartado escribiremos algunos consejos que podrían llegar a ser utiles en un futuro.

Devolver un objeto literal

A veces necesitamos devolver objetos literales y si utilizamos la sintaxis de **fat arrow** puede dar problemas por ejemplo:

```
var foo = ()=>{  
  bar: 123  
};  
console.log(foo);
```

Esto al ser una función anónima, en realidad, **no** nos devuelve el objeto literal sino que se muestra así

```
function () {  
  bar: 123  
};
```

Mientras que si lo hacemos de la siguiente forma podremos obtener un objeto literal con una función que utilice **fat arrow**

```
var foo = ()=>({  
  bar: 123  
});
```

Esto nos devolverá:

```
function () { return ({  
  bar: 123  
}); };
```

Clases estáticas

Una característica general de otros lenguajes de programación es la palabra `static` para incrementar la duración de una variable (no en lo referente a el ambito de la variable), Aquí tenemos un ejemplo de uso del lenguaje de programación `c` :

```
void called() {
    static count = 0;
    count++;
    printf("Called : %d", count);
}

int main () {
    called(); // Called : 1
    called(); // Called : 2
    return 0;
}
```

Como javascript o TypeScript no tiene funciones o clases estáticas existe una manera de hacerlo la manera sería la siguiente:

```
const {called} = new class {
    count = 0;
    called = () => {
        this.count++;
        console.log(`Called : ${this.count}`);
    }
};

called(); // Called : 1
called(); // Called : 2
```

Métodos de inicialización estáticos

Como **TypeScript** al igual que javascript no pueden crear clases estáticas si que se pueden crear método estático de inicialización.

```
class MyClass {  
    static initialize() {  
        // Initialization  
    }  
}  
MyClass.initialize();
```

¿Quiénes somos?



Emmanuel Valverde Ramos - [@evrtrabajo](#)

- Programador especializado en desarrollo web
- Conocimientos en:
 - PHP
 - Java
 - C
 - PL/SQL
 - HTML5
 - CSS3
 - JQuery
 - Javascript
 - TypeScript
 - XML
 - XPath
 - Ingeniería inversa
 - Web Crawling
 - Web Scraping



Pedro Hernández-Mora de Fuentes

- Programador especializado en desarrollo web
- Conocimientos en:
 - **PHP**
 - **Java**
 - **PL/SQL**
 - **HTML5**
 - **CSS3**
 - **JQuery**
 - **Javascript**
 - **TypeScript**
 - **XML**
 - **XPath**
 - **Ingeniería inversa**
 - **Web Crawling**
 - **Web Scraping**



El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la Programación Java, C/C++/C#, Visual.Net, SQL, Python, Javascript, Oracle, Algoritmos, CSS, Desarrollo Web, Joomla, jquery, Ajax y Mucho Mas...

VISITA

www.detodoprogramacion.org

www.detodopython.com

www.gratiscodigo.com



GRATISCODIGO.COM

Usa, lo que ya se creó...



Detodo
Python.com