

# Introduction to TypeScript



# Our Team



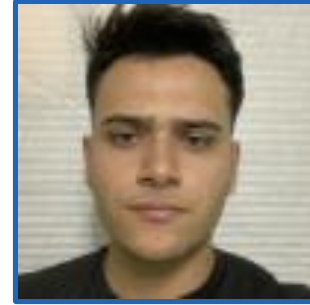
**Eric Bermúdez Hernández**

eric.bermudez.16@ull.edu.es



**Adrián Alejandro  
Padrón López**

adrian.padron.30@ull.edu.es



**Diego Rodríguez  
Martín**

diego.rodriguez.28@ull.edu.es



# Index

**01**

**Introduction**

**02**

**Installation**

**03**

**Basic Concepts**

**04**

**Remarkable  
Features**

**05**

**OOP**

**06**

**Interfaces**

**07**

**Modules**

**08**

**Bibliography**



01

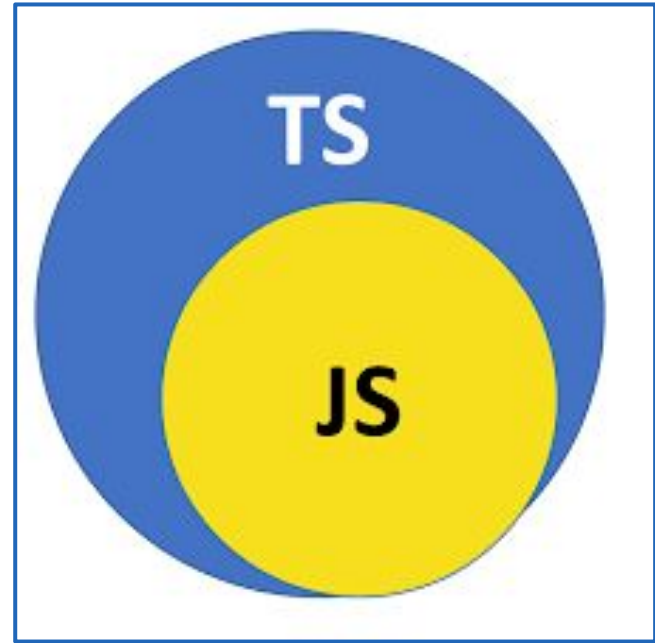
# Introduction



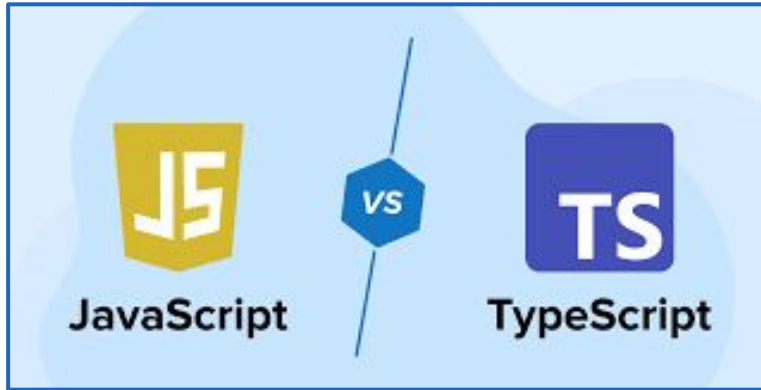
# What is TypeScript?

TypeScript is a programming language that comes as a superset of JavaScript. This means that any valid JavaScript code is also valid TypeScript code.

However, TypeScript adds additional features and tools that improve the development experience, especially on large, complex projects.

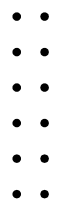


# Main differences



The main differences between Typescript and Javascript are:

1. Typing.
2. Object-oriented programming.
3. Improved development tools.
4. Greater scalability and maintainability.



# Benefits of using TypeScript



1. Early error detection.
2. Better readability and documentation.
3. Greater productivity.
4. Greater security.
5. JavaScript support.



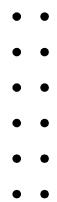


02

# Installation







# ¿What do we need?

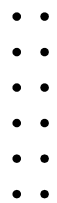
1. Install Node.js and npm/pnpm



Node version must be greater than **14.17**

2. Install a TypeScript compiler

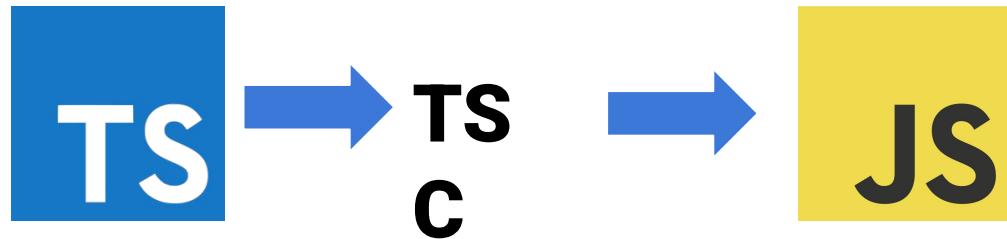




# We have two options to compile TS!



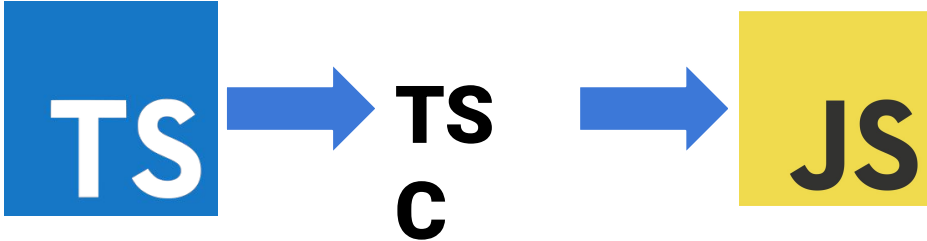
1. With 'tsc'.

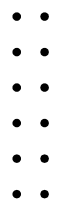


2. With 'ts-node'.



⋮ **First one,**  
⋮ **'tsc'**  
⋮

- Official TS compiler 
- Translates TS code into JS code, generating a new .js file
- Check static types for errors before executing the code



## How to install 'tsc'

→ Execute the following command:

```
npm install -g typescript
```

→ Check the version with:

```
tsc -v
```



⋮ **Second one,**  
⋮ **'ts-node'**  
⋮



- Compiles and runs TypeScript code directly
- More comfortable than tsc



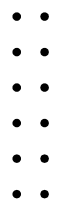
# ⋮ How to install ⋮ 'ts-node' ⋮

→ Execute the following command:

```
npm install -g ts-node
```

→ Check the version with:

```
ts-node -v
```



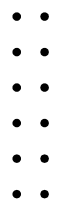
## Starting a TS project

We can generate the package.json using the following command:

```
npm init -y
```

→ But now we need a configuration file for TS!





## **tsconfig file**

- **Configuration file for TypeScript**
- **Defines how to compile the code**
- **We can customize this file with many options**
- **[A tipic TScnfig.json](#)**



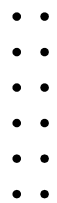




03

# Basic Concepts

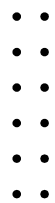




# Types

- string
- number
- boolean
- array
- object
- tuple: 2 or more elements of different type
- enum: list of constants





# Types


- **any**: lets every type. Avoid using: breaks the strongly typed feature
- **unknown**: similar like any but it can't be assigned to any type unlike any.
- **never**: it never should be a type. In a function that returns a never value, we are specifying that function should never return anything.
- **void**: no value





# Variable Declaration



- `let`
  - `const`
  - NEVER use `var` to declare a variable
- 

# Why NOT using *var*?

- `var` has function scope instead of block scope.
- This means it ignores the block (like `if {}`, `for {}`) in which it is wrapped.

```
if (true) {  
  var message: string = 'Hello World';  
}  
  
console.log(message); // Hello World
```

# Why NOT using *var*?

- `var` allows redeclarations
- This makes the code very confusing, also causing inconsistencies and unexpected errors

```
var userName: string = "Diego";  
var userName: string = "Adrian";  
console.log(userName);
```

# Why NOT using *var*?

- `var` implies hoisting what means the declarations go to the start of the scope without initialization.
- This may produce confusing errors, because the value will be undefined instead of throwing an error

```
console.log(greeting); // undefined
var greeting: string = 'Hello World';
```

## Conditionals: if-else

```
// if-else statement
let age: number = 18;
if (age >= 18) {
  console.log("You are an adult");
} else {
  console.log("You are a minor");
}
```




# Conditionals: Switch Statement

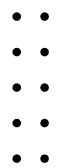
```
// switch statement
let color: string = 'red';
switch (color) {
  case 'red':
    console.log("The color is red");
    break;
  case 'blue':
    console.log("The color is blue");
    break;
  default:
    console.log("Color not recognized");
}
```



# Conditionals: Ternary Operator

```
// ternary operator
let height: number = 180; // height in centimeters
let message = height >= 185 ? "You are a tall person" : "You are a medium/short height person";
console.log(message);
```





# Loops: for, for of



```
// Classic for loop
console.log("Classic for loop");
for (let i: number = 0; i < 5; i++) {
  console.log(i);
}
```

```
// For of loop
console.log("For of loop");
let fruits: string[] = ["Apple", "Pear", "Grape"];
for (let fruit of fruits) {
  console.log(fruit);
}
```



# Loops: for in, forEach

```
// For in loop
console.log("For in loop");
let person = { name: "John", age: 25 };
for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

```
// Foreach loop
console.log("Foreach loop");
let numbers: number[] = [1, 2, 3];
numbers.forEach(num => console.log(num));
console.log();
```

# Loops: while

```
// While loop
console.log("While loop");
let counter: number = 0;
while (counter < 3) {
    console.log(counter);
    counter++;
}
```

# Functions: Declared Functions

```
function greet(name: string): string {  
  return `Hola, ${name}`;  
}  
  
console.log(greet("Ana"));
```



# Functions: Anonymous Functions



```
const sum = function (operand1: number, operand2: number): number {  
  return operand1 + operand2;  
};  
  
console.log(sum(3, 4));
```

# Functions: Arrow Functions

```
// Arrow function that calculates factorial of a number
const factorial = (number: number): number => {
  if (number === 0) {
    return 1;
  }
  return number * factorial(number - 1);
};
```



# Functions: Optional Parameters

- Important: Optional parameters must be always after the required ones

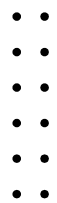
```
// Function with optional parameter
function multiplyTwoOrThreeNumbers(operand1: number, operand2: number, operand3?: number): number {
  if (operand3) {
    return operand1 * operand2 * operand3;
  }
  return operand1 * operand2;
}
```



04

# Remarkable Features





# Type Assertions

- Allows you to change the type of an specified value, to be treated like that
- You can use `as` or `< >` as well

```
let a: typeA;  
let b = a as typeB;
```

```
let a: typeA;  
let b = <typeB>a
```



```
const element = document.querySelector('input[type="text"]');  
console.log(element.value);
```

- element is an Element type value
- Does not have *value* property



```
const input = element as HTMLInputElement;
```

```
console.log((<HTMLInputElement>element).value);|
```



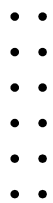
# Type Assertion Errors

- Compile-Time Errors

```
let price = '9.99';  
let numberPrice = price as number; // error
```

- Runtime errors: if element is not an input element

```
let element = document.querySelector('#name');  
let input = element as HTMLInputElement;  
console.log(input.value.length);|
```



# Non Null Assertion

- `!` operator
- Avoiding unnecessary null and undefined checks in our code.
- Only use this when we definitely know the variable or expression can't be null or undefined



From this

```
function duplicate(text: string | null) {  
  if (text === null || text === undefined) {  
    text = "";  
  }  
  return text.concat(text);  
}
```


To this

```
return text!.concat(text!);
```



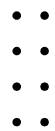
# Optional Chaining



- Access an object property or calls a function
  - If is undefined or null -> short circuits and evaluates to undefined instead of throwing an error.
  - Shorter and simpler expressions when accessing chained properties when the possibility exists that a reference may be missing.
- 



const nestedProp = obj.first?.second;



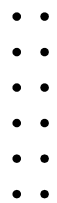
# Optional Chaining

```
const nestedProp = obj.first && obj.first.second; // Before
```

```
const nestedProp = obj.first?.second; // After
```

```
let x = foo?.bar.baz()
```

```
let x = foo === null || foo === undefined ? undefined : foo.bar.baz()
```



# Optional Chaining with Methods



- If `customMethod` is not found, return `undefined`
- If `someInterface` may be also `null` or `undefined`, use `?.` at this point as well.

```
const result = someInterface?.customMethod?.();
```

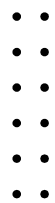


# Optional Chaining: Invalid Cases

- Assign to the result of an optional chaining expression
- The constructor of new expressions cannot be an optional chain

```
const object: Object = {  
  size: 2  
};  
object?.size = 1; // ERROR
```

```
new Number(1)?.toFixed(); // ERROR
```



# Nullish coalescing operator




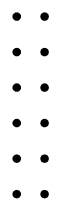
- If left is null or undefined -> return right. Otherwise -> return left
- It is used to give default values.
- “Special case” of OR (||) operator. While OR operator returns right operand if left operand is any falsy value, ?? operator returns it only if it's a null or undefined value.





# Nullish coalescing operator

- Usage: if you treat 0 or "" as valid values, you may use coalescing operator instead because these are "falsy" values.
  - It is important that it is not possible to combine it with && and || operators without parentheses.
- 



# Nullish coalescing operator



```
const myText = "";  
  
const noFalsyText = myText || "default text";  
console.log(noFalsyText); // "default text"  
  
const preservingFalsyText = myText ?? "default text";  
console.log(preservingFalsyText); // ""
```





# Relationship with Optional Chaining Operator

```
const foo = {
  someProperty: "hi"
};

console.log(foo.someProperty?.toUpperCase() ?? "default text"); // "HI"
console.log(foo.someSecondProperty?.toUpperCase() ?? "default text"); // "default text"
```

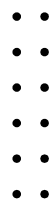


05

OOP







# ¿What is a class?



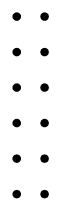
- Is a template to create objects
- Defines the properties(data) and methods(actions) for objects
- Used to organize and structure the code



# ⋮ Differences between C++ and TS ⋮ classes

→ In TS:

- Creates constructor with the keyword 'constructor'
- 'Readonly' properties
- You cannot overload constructors
- Getters and Setters are created with the keyword 'get' and 'set' respectively



# Differences between C++ and TS classes




- Public visibility by default for everything
- 'Extends' keyword for inheritance
- Abstract classes with keyword 'abstract' both for the definition of the class and for the methods
- Properties at the beginning of the class (Google style)
- Classes do not end with semicolon ';'





# ⋮ Differences between C++ and TS ⋮ classes

- All members of a TypeScript class must specify their access level using public, private, or protected.  
Example: [Visibility in TS classes](#)
  - Methods cannot be defined in a class and implemented outside of it, they must be fully implemented within the class.
- 

# Parameter Properties

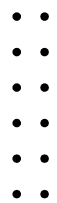
- In TypeScript, adding an access modifier to a constructor parameter automatically creates a class property with the same name and assigns it the given value.

→ Example with parameter properties:

[With parameter properties](#)

→ Example without parameter properties:

[Without parameter properties](#)



# Keyword this

- Is a reference to the current object on which the code is being executed
- Inside a TypeScript class, it is used to access properties and methods of the actual object.

```
class Car {  
  brand: string;  
  // ... more properties+  
  
  constructor(brand: string) {  
    this.brand = brand;  
  }  
}
```



# Read only properties

- This type of properties cannot be changed!



Example of this:

[readonly properties example](#)

- There are differences between readonly and const properties

# ⋮ Differences between readonly and const

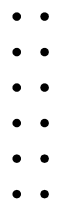
## readonly

- When we assign a value we cannot modify it
- Is used for properties within the members of a class

## const

- We need to assign a value when we declare it
- It is used for variables





# Constructor overloading

- You can only implement one constructor
- But we can simulate this with optional parameters in the constructor

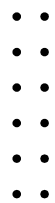
→ Example:

[Simulating constructor overloading](#)



# Getters and Setters

```
class Player {  
    constructor(private name: string) {}  
  
    get playerName(): string {  
        return this.name;  
    }  
  
    set playerName(newValue: string) {  
        this.name = newValue;  
    }  
}  
  
let playerOne = new Player('Rachel');  
playerOne.playerName = 'Alice';
```



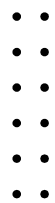
# Static members

- Useful when you need functionalities that are related with the class and not with a specific object.  
Some common examples are:
  - Constants: Values that are shared by all instances of a class.
  - Counters: Variables that keep track of the number of instances created.

→ Example of a class with a static member:

[Static members](#)





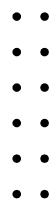
# Static members

- Useful when you need functionalities that are related with the class and not with a specific object.  
Some common examples are:
  - Constants: Values that are shared by all instances of a class.
  - Counters: Variables that keep track of the number of instances created.

→ Example of a class with a static member:

[Static members](#)





# Class inheritance

- We need the keyword 'extends' for class inheritance

→ Example of simple inheritance:

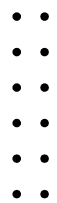
## Simple Inheritance

- TS not supports multiple inheritance, but it can be simulated with interfaces

→ Example of a simulation of multiple inheritance:

## Simulation multiple inheritance

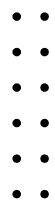




## Keyword 'Super()'

- The `super()` keyword in TypeScript calls the parent class constructor, ensuring proper initialization before adding subclass functionality.
- It is required to call `super()` in the constructor of a subclass before using the keyword 'this'
- If the parent class has a constructor with parameters, `super()` should receive those values.
- `Super.method()` can be used to call methods of the parent class. Use example of 'Super()': [Super\(\) use example](#)



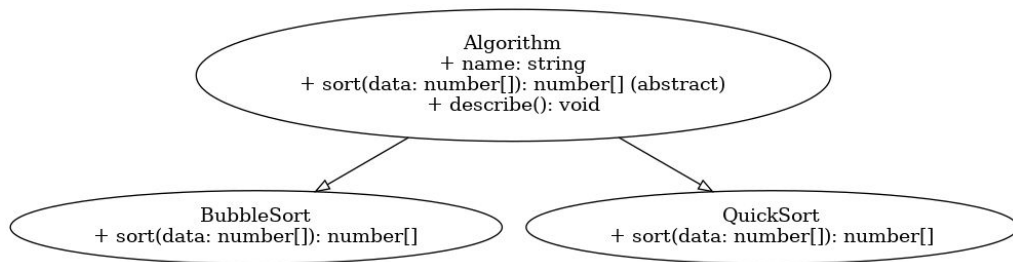


# Abstract classes

- Templates for other classes
- They cannot be instantiated directly
- They are used when several classes share a common structure, but each one needs to implement its own version of certain methods



# Abstract classes



- We use the keyword 'abstract' before the keyword 'class'
  - They are defined with abstract and have no implementation in the base class. They must be implemented in the derived classes.
  - An abstract class can have concrete methods and properties that will be inherited by its subclasses.
- Example of an abstract class: [Abstract classes](#)



# Templates in TS classes

- Templates allow classes to work with multiple data types without specifying one beforehand, making code more reusable and flexible

```
class Pair<T, U> {  
  private first: T;  
  private second: U;  
  
  constructor(first: T, second: U) {  
    this.first = first;  
    this.second = second;  
  }  
  // Method to display the pair  
  display(): void {  
    console.log(`First: ${this.first}, Second: ${this.second}`);  
  }  
}  
  
// Creating instances with different types  
let numberPair = new Pair<number, number>(5, 10);  
let stringNumberPair = new Pair<string, number>("Age", 30);  
// Using the display method  
numberPair.display();           // Output: First: 5, Second: 10  
stringNumberPair.display();     // Output: First: Age, Second: 30
```



06


# Interfaces





# Interfaces

Interfaces in TypeScript are a fundamental tool for defining the shape that objects in your code should have. They act as contracts that specify what properties and methods the objects that implement them must have. It should be noted that everything that is within an interface is public.



```
interface Person {  
  name: string;  
  age: number;  
  sex: string;  
  nationality: string;  
}
```

# Interface extension

```
interface BasicAddress {  
  name?: string;  
  street: string;  
  city: string;  
  country: string;  
  postalCode: string;  
}
```

```
let address: AddressWithUnit = {  
  name: "John Doe",  
  street: "123 Main Street",  
  city: "Springfield",  
  country: "USA",  
  postalCode: "12345",  
  unit: "456",  
};
```

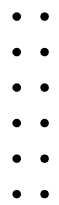
```
interface AddressWithUnit extends BasicAddress {  
  unit: string;  
}
```

# Intersection types

```
interface Colorful {  
  color: string;  
}  
  
interface Circle {  
  radius: number;  
}  
  
type ColorfulCircle = Colorful & Circle;
```

```
let circle: ColorfulCircle = {  
  color: "blue",  
  radius: 42  
};  
draw(circle);
```

```
function draw(circle: ColorfulCircle) {  
  console.log(`Drawing a ${circle.color} circle with radius ${circle.radius}`);  
}
```



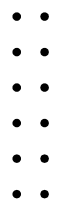
# Interface extension VS Intersection type

We observe two ways of combining types that are similar, but are actually subtly different. The main difference between the two is how conflicts are handled.

In case of interface extension. TypeScript will try to merge them if the properties are compatible. If the properties are not compatible (that is, they have the same property name but different types), TypeScript will raise an error.

In the case of intersection types, properties with different types will be merged automatically. When the type is used later, TypeScript will expect the property to satisfy both types simultaneously, which can produce unexpected results.





# Abstract classes

- Mix between already implemented methods and abstract methods.
- A class can only inherit / extend a single abstract class.
- They have all available types of visibility.
- They can have a constructor.



# Interface s



- They are completely abstract.
- A class can implement multiple interfaces.
- They act as contracts.
- Everything is public.



07

# Modules



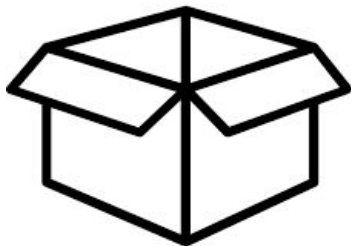


# What is a module?

Modules in TypeScript are a fundamental tool for organizing and structuring code in projects of any size. They allow the code to be divided into logical and reusable units, which makes it easier to understand, maintain and scalable.

Reasons why you should use modules:

- Better code organization.
- Avoid name collisions.
- Code reuse.
- Better performance.



# Types of modules

## CommonJS

- First widely used module system in JavaScript
- Use “require” to import.
- Use “module.exports” to export.
- Module loading is **synchronous**.



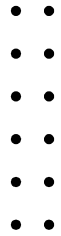
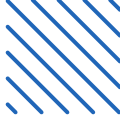
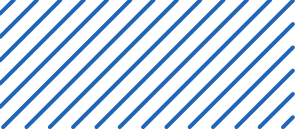
CJS

## ES Modules

- Standard for modern modules.
- Use “import” to import.
- Use “exports” to export.
- Module loading is **asynchronous**.

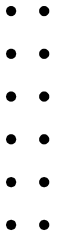


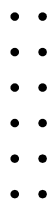
ESM



08

# Bibliograph y





Tipos de datos en TypeScript. Nicolás F. Ormeño Rojas - Medium. Disponible en:  
<https://normeno.medium.com/tipos-de-datos-en-typescript-d67482f2da6c>

TypeScript Best Practices 2021. Warki Ringoda - Medium. Disponible en:  
<https://medium.com/@warkiringoda/typescript-best-practices-2021-a58aee199661>

Type Casting. TypeScript Tutorial. Disponible en: <https://www.typescripttutorial.net/typescript-tutorial/type-casting/>

Non-Null Assertion Operator. Learn TypeScript. Disponible en:  
<https://learntypescript.dev/07/l2-non-null-assertion-operator>

Nullish Coalescing Operator (??). MDN Web Docs. Disponible en:  
[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish\\_coalescing](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing)

TypeScript 3.7 Release Notes. TypeScript Official Website. Disponible en:  
<https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html>

Optional Chaining (?). MDN Web Docs. Disponible en:  
[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional\\_chaining](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining)



• • Hoisting. MDN Web Docs. Disponible en: <https://developer.mozilla.org/es/docs/Glossary/Hoisting>

• •

• • Explain about noImplicitAny in TypeScript. GeeksforGeeks. Disponible en:  
• • <https://www.geeksforgeeks.org/explain-about-noimplicitany-in-typescript/>

tsconfig.json. TypeScript Official Website. Disponible en:  
<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

¿Qué es TypeScript vs JavaScript?. Profile.es. Disponible en: <https://profile.es/blog/que-es-typescript-vs-javascript/>

TypeScript vs JavaScript: Similitudes y diferencias. Imagina Formación. Disponible en:  
<https://imaginaformacion.com/tutoriales/typescript-vs-javascript-similitudes-y-diferencias>

Entendiendo ES Module y CommonJS: diferencias clave y compatibilidad. NelkoDev. Disponible en:  
<https://nelkocodev.com/blog/entendiendo-es-module-y-commonjs-diferencias-clave-y-compatibilidad/>

Modules. TypeScript Official Website. Disponible en:  
<https://www.typescriptlang.org/docs/handbook/2/modules.html>

tsconfig. TypeScript Official Website. Disponible en: <https://www.typescriptlang.org/tsconfig/>

CommonJS vs ESM. Lenguaje JS. Disponible en: <https://lenguajejs.com/nodejs/fundamentos/commonjs-vs-esm/>



# THANKS!

**Do you have any questions?**

Eric Bermúdez Hernández [eric.bermudez.16@ull.edu.es](mailto:eric.bermudez.16@ull.edu.es)

Adrián Alejandro Padrón López [adrian.padron.30@ull.edu.es](mailto:adrian.padron.30@ull.edu.es)

Diego Rodríguez Martín [diego.rodriquez.28@ull.edu.es](mailto:diego.rodriquez.28@ull.edu.es)

