

Introduction to TypeScript

What is TypeScript?



- High-level programming language
- JavaScript superset
- Compiled to interpreted language
- Strongly typed language

What is TypeScript?

- TypeScript is a superset of JavaScript
- Everything in TS is JS
- Not everything in JS is TS



What is TypeScript?

- TypeScript can't be interpreted
- Can be compiled into JS
- Only then can it be interpreted



What is TypeScript?

- Strongly typed language
- Safer
- Less prone to errors



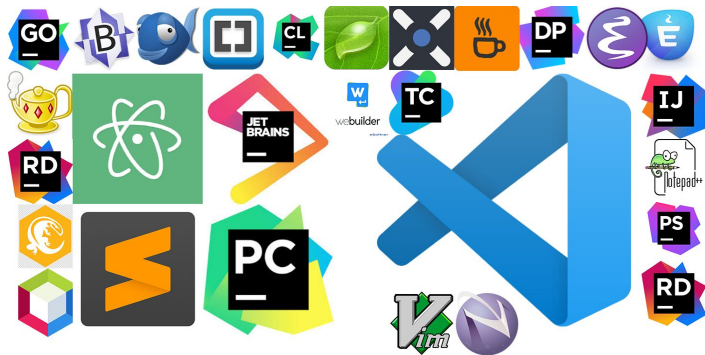
Setting up TypeScript

What you need:

1- Node.js

2- TypeScript compiler

3- Code editor



Setting up TypeScript

Installing Curl:

```
$ sudo apt-get install curl
```

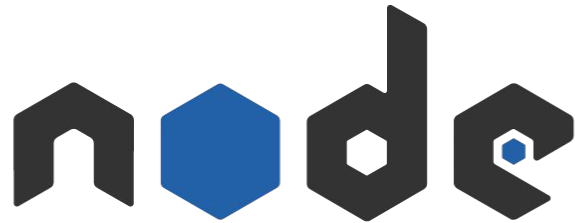
Installing Node.js and Npm:

```
$ curl -sL https://deb.nodesource.com/setup_18.x | sudo -E bash -  
$ sudo apt-get install -y nodejs
```

Setting up TypeScript

Two types of **TypeScript** compilers

TypeScript
Compiler



Setting up TypeScript

TypeScript compiler:



- Compiles TS code into JS code
- Can be then interpreted with Node.js

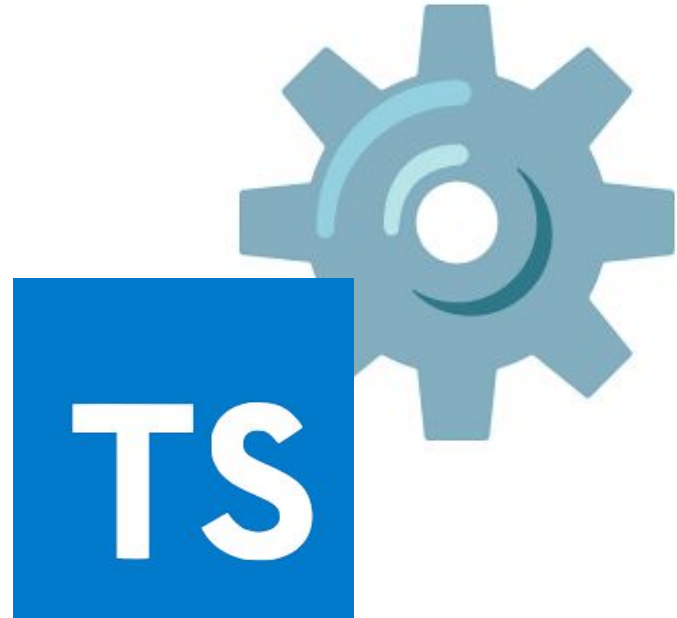


```
npm install -g typescript
```

Setting up TypeScript

Setting up a **TypeScript** compiled project:

```
npx tsc --init (tsconfig.json)  
npx tsc
```



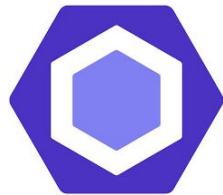
TSconfig.json:

- Is the configuration file for the compiler.
- Sets all the options before compiling (npx tsc)
- Example basic tsconfig.json:



```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es6",
    "noImplicitAny": true,
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist",
    "baseUrl": ".",
    "paths": {
      "**": [
        "node_modules/*"
      ]
    }
  },
  "include": [
    "src/**/*"
  ]
}
```

ESLint para TypeScript:



- ESLint support typescript code.
- For setting in the project:

```
npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin  
npx eslint --init
```

- Generates **.eslint.json**
- Add to **package.json**:

```
"lint": "eslint . --ext .ts"
```

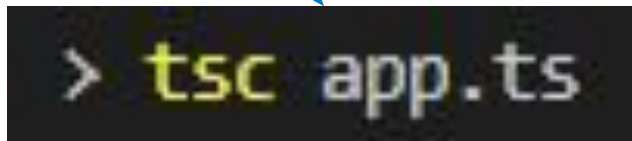
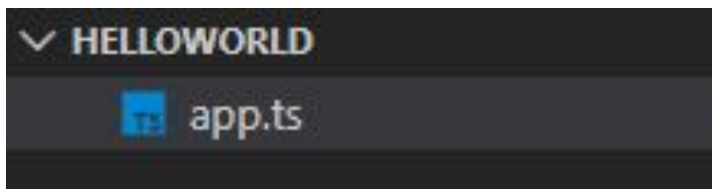
- Execute the Linter:

```
npm run lint
```

```
{  
  "env": {  
    "browser": true,  
    "commonjs": true,  
    "es2021": true  
  },  
  "overrides": [  
  ],  
  "parser": "@typescript-eslint/parser",  
  "plugins": [  
    "@typescript-eslint"  
  ],  
  "extends": [  
    "eslint:recommended",  
    "plugin:@typescript-eslint/eslint-recommended",  
    "plugin:@typescript-eslint/recommended"  
  ],  
  "rules": {  
    "no-console": 2  
  }  
}
```

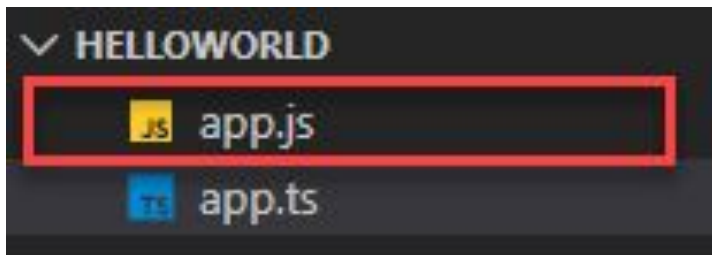
Setting up TypeScript

Using the **TypeScript** compiler:



```
> tsc app.ts
```

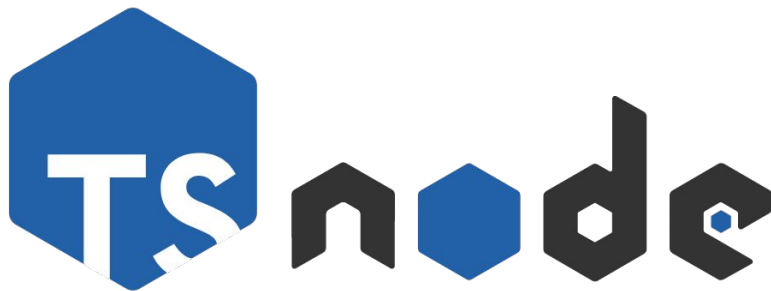
A terminal window showing the command to compile the TypeScript file.



TypeScript
Compiler

Setting up TypeScript

TypeScript node:


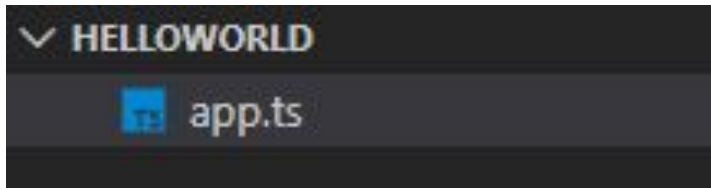


- Recommended option
- Compiles automatically
- Appears to interpret **TypeScript** directly

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

Setting up TypeScript

Using **TypeScript** node:



```
> ts-node app.ts
```



Hello World example:

TypeScript
Compiler

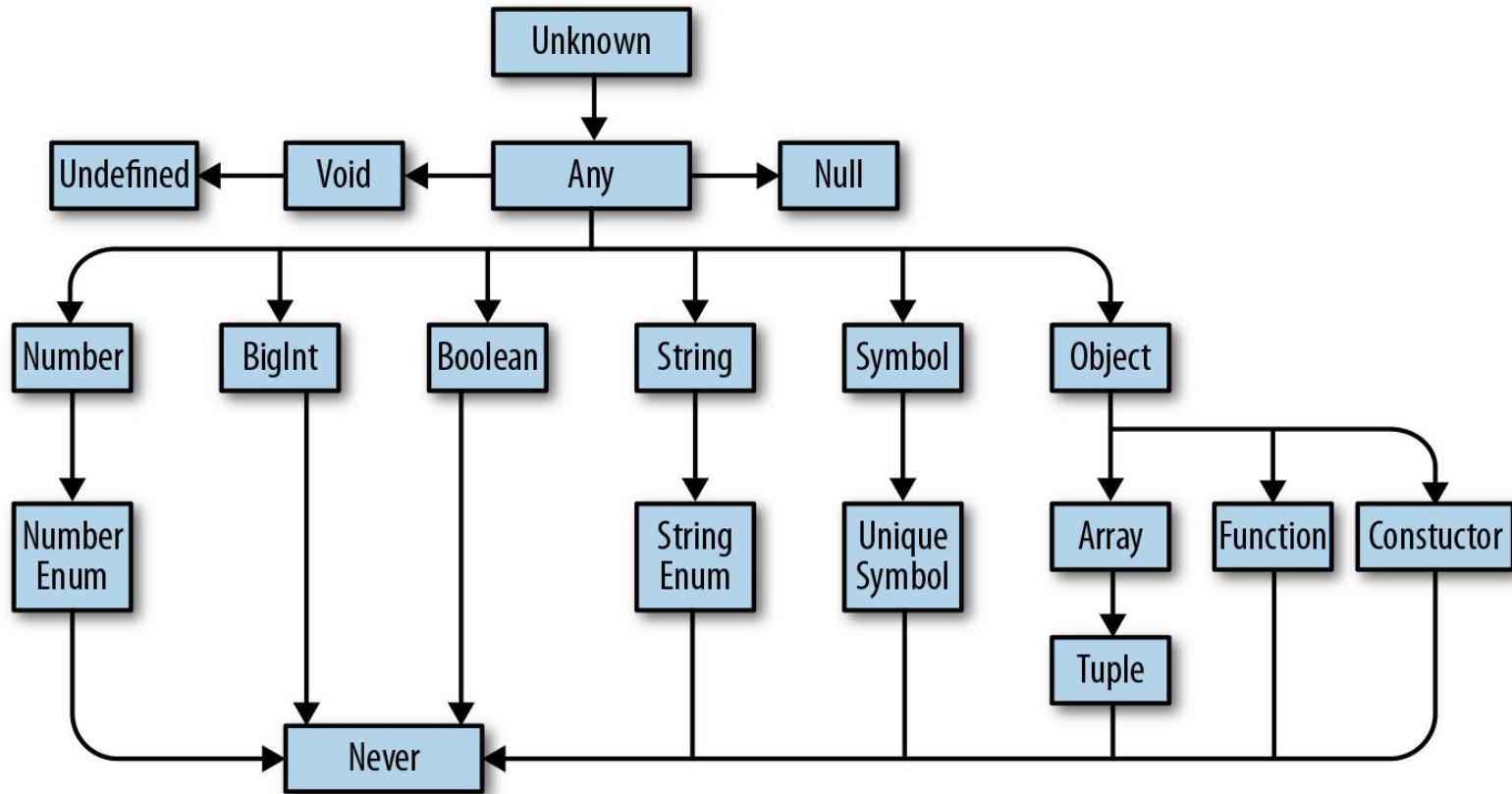


Declaring types:

```
let variable: string = "hello";  
let anotherVariable: number = 5;  
let arrayVariable: number[] = [1, 2, 7];
```

```
let variable: number | string  
variable = 10; // Fine  
variable = "Hello" // Also fine  
variable = false; // Nope
```

Types in TypeScript:



Number:

```
let decimal: number = 6;  
let floatingPoint: number = 1.19982  
let hex: number = 0xf00d;  
let binary: number = 0b1010;  
let octal: number = 0o744;
```

- Floating point
- Base 2, 8, 10...

BigInt:

```
let big: bigint = 100n;
```

- Bigger than number

Boolean:

```
let trueBoolean: boolean = true;  
let falseBoolean: boolean = false;
```

- Either true or false

String:

```
let age: number = 15;
```

```
let color: string = "blue";
```

```
let sentence: string = `Hello, I'm ${age} years old and my favourite color is ${color}`
```

Array:

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

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

- Values have to be the same type

Tuple:

```
let tuple: [number, string, boolean] = [10, "hello", true];
```

- Can have different types
- Types need to be declared

Unknown:

```
1 let notSure: unknown = 4;  
2 notSure = "maybe a string instead";  
3  
4 // OK, definitely a boolean  
5 notSure = false;
```

- Can be anything
- Can change at any moment

Any:

```
let anyVariable: any = 4;  
// OK, someProperty might exist at runtime  
anyVariable.someProperty();  
// OK, toFixed exists (but the compiler doesn't check)  
anyVariable.toFixed();
```

```
let unknownVariable: unknown = 4;  
unknownVariable.toFixed();
```

```
let unknownVariable: unknown
```

```
'unknownVariable' is of type 'unknown'
```

- Same as unknown
- But allows you to check for properties

Void:

```
function voidFunction(): void {  
  console.log("This function returns nothing");  
}
```

- Nothing
- Used for variables without value like function returns

Null and Undefined:

```
// Not much else we can assign to these variables!  
let u: undefined = undefined;  
let n: null = null;
```

- Null and Undefined have their own type

Never:

```
// Function returning never must not have a reachable end point
```

```
function error(message: string): never {  
  throw new Error(message);  
}
```

```
// Inferred return type is never
```

```
function fail() {  
  return error("Something failed");  
}
```

```
// Function returning never must not have a reachable end point
```

```
function infiniteLoop(): never {  
  while (true) {}  
}
```

– Represents values
that never happen

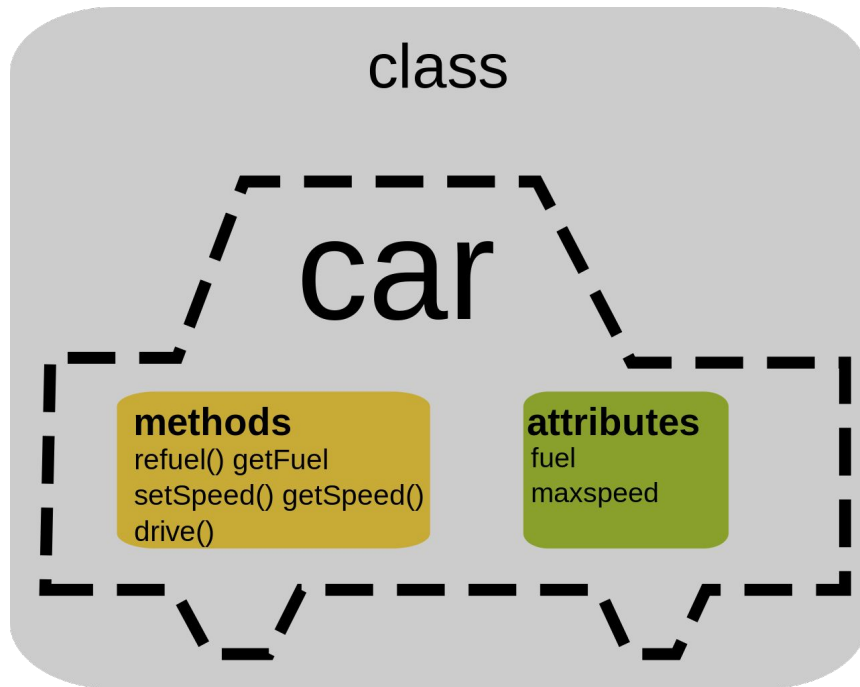
Optional/Default parameters:

Functions in TS can have optional and default parameters

```
function optionalParameterFunction(neededParameter: number, optionalParameter?: number): void {  
    // some code  
    return;  
}  
  
function defaultParameterFunction(neededParameter: string, defaultParameter: number = 10): void {  
    return;  
}
```

Classes:

- TS lets you create fully fledged classes
- You can specify types
- You can define attributes and methods.
- There's class inheritance.
- You can define abstract classes & methods.



Classes:

```
class Person {
  private firstName: string;
  private readonly lastName: string;
  private age: number;

  constructor(firstName: string, lastName: string, age: number) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

  public getFullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }

  public changeLastName(lastName: string): void {
    this.lastName = lastName; // This will give an error
    return;
  }

  protected getAge(): number {
    return this.age;
  }
}

const person = new Person("Pedro", "Piqueras", 67);
console.log(person.getFullName());
```

Example of a class:

Classes: Static Method

- The static members are accessed using the class name
- Don't need to instantiate an object to use it.

```
class Circle {  
    static pi: number = 3.14;  
    static calculateArea(radius: number) {  
        return this.pi * radius * radius;  
    }  
}  
  
Circle.pi; // returns 3.14  
Circle.calculateArea(5); // returns 78.5
```


Classes: Inheritance

↓ Example of inheritance:

```
class Employee extends Person {  
  private jobTitle: string;  
  
  constructor(firstName: string, lastName: string, age: number, jobTitle: string) {  
    // Calls the constructor of the Person class:  
    super(firstName, lastName, age);  
    this.jobTitle = jobTitle;  
  }  
  
  describe(): string {  
    return `I'm a ${this.jobTitle}, my name is ${super.getFullName()} and I'm ${super.getAge()}`;  
  }  
}
```

Classes: Overrides

Method overriding:

```
class Employee extends Person {  
    private jobTitle: string;  
  
    constructor(firstName: string, lastName: string, age: number, jobTitle: string) {  
        // Calls the constructor of the Person class:  
        super(firstName, lastName, age);  
        this.jobTitle = jobTitle;  
    }  
  
    public changeLastName(lastName: string): void {  
        console.log("I'm sorry but changing the last name of an employee is not allowed");  
        return;  
    }  
}
```

Classes: Abstracts

```
abstract class Vehicle {
  protected kilometers: number;
  private brand: string;
  private model: string;

  constructor(kilometers: number, brand: string, model: string) {
  }

  abstract getQuality(): number;

  get carInfo(): string {
    return `${this.brand} ${this.model}, ${this.kilometers}km`;
  }
}

let myCar = new Vehicle(87000, 'Volkswagen', 'Tiguan'); // You can't create instances of abstract classes
```

Abstract classes:

```
class Car extends Vehicle {

  constructor(kilometers: number, brand: string, model: string) {
    super(kilometers, brand, model)
  }

  getQuality(): number {
    return this.kilometers * 0.1;
  }
}

let volksCar = new Car(87000, 'Volkswagen', 'Tiguan');
```

Data Structuring (Types):

- Create your aliases for complex types.
- Reuse them in all your code.
- Better for clean code.
- restricts the object type.

```
type operator = {  
    firstValue: number;  
    secondValue: number;  
}
```

Data Structuring(Interface)

- Create your aliases for complex types.
- Reuse them in all your code.
- Better for clean code.
- restricts the object type.

```
interface operator | {  
  firstValue: number;  
  secondValue: number;  
}
```

Data Structuring(Interface/types)

```
interface operator | {  
  firstValue: number;  
  secondValue: number;  
}
```

===

```
type operator = {  
  firstValue: number;  
  secondValue: number;  
}
```



Data Structuring(Interface/types)

```
interface operator | {  
  firstValue: number;  
  secondValue: number;  
}
```

~~=~~

```
type operator = {  
  firstValue: number;  
  secondValue: number;  
}
```



Data Structuring(Interface)

- Interfaces can **Extend** the Data structure in any moment, type is a strict data structure.

```
interface Door {  
  width: number;  
  height: number;  
}  
  
interface Door {  
  material: string;  
}  
  
let windowOne :Door = { width: 12, height: 12 , material: "acero"};  
  
console.log(windowOne);
```


Generics (Templates):

- They don't exist in JavaScript
- They are a “template” for code.
- They serve to optimize the code, avoiding duplicates in functions.
- Safer than using an ‘any’ parameter for the function

```
generics.ts / ...  
//generic function  
function display<template>(value: template): template {  
  console.log(value);  
  return value;  
}
```

Generics (for functions):

Example:

```
function getRandomElement<type>(items: type[]): type {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}  
  
let array: number[] = [3, 5, 6, 8]  
let result = getRandomElement<number>(array);  
console.log(result);
```

Generics (for interfaces):

Example:

```
//Generic Interface
```

```
...
```

```
interface GenericInterface<T, U> {
```

```
    key: T;
```

```
    value: U;
```

```
}
```

```
let kv1: GenericInterface<number, string> = { key:1, value:"P" };
```

```
let kv2: GenericInterface<number, number> = { key:1, value:12345 };
```

Generics (for classes):

Example:

```
class GenericClass<T,U> {  
    valOne: T;  
    valTwo: U;  
  
    constructor(valOne: T, valTwo: U) {  
        this.valOne = valOne;  
        this.valTwo = valTwo;  
    }  
}  
  
const objectOne = new GenericClass<number,number>(1,1);  
const objectTwo = new GenericClass(2,"pepe");
```