TypeScript

TypeScript is JavaScript with added syntax for types.

What is TypeScript?

TypeScript is a syntactic superset of JavaScript which adds **static typing**.

This basically means that TypeScript adds syntax on top of JavaScript, allowing developers to add **types**.

TypeScript being a "Syntactic Superset" means that it shares the same base syntax as JavaScript, but adds something to it.

Why should I use TypeScript?

JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.

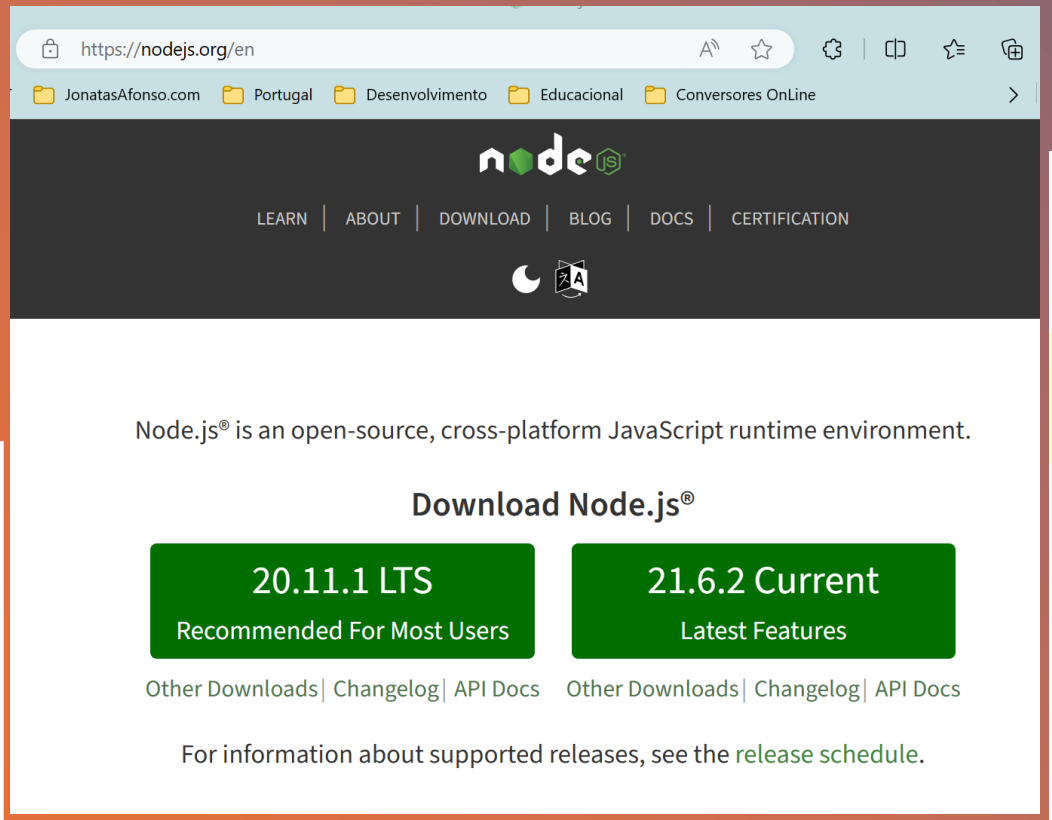
In JavaScript, function parameters and variables don't have any information! So developers need to look at documentation, or guess based on the implementation.

TypeScript allows specifying the types of data being passed around within the code, and has the ability to report errors when the types don't match.

For example, TypeScript will report an error when passing a string into a function that expects a number. JavaScript will not.

TypeScript uses compile time type checking. Which means it checks if the specified types match **before** running the code, not **while** running the code.

Instaling the Compiler.



Installing the Compiler

TypeScript has an official compiler which can be installed through npm.

Learn more about npm, and how to get started here: What is npm?

Within your npm project, run the following command to install the compiler:

```
npm install typescript --save-dev
```

Which should give you an output similar to:

```
added 1 package, and audited 2 packages in 2s
found 0 vulnerabilities
```

The compiler is installed in the `node_modules` directory and can be run with: `npx tsc`.

```
npx tsc
```

Which should give you an output similar to:

```
Version 4.5.5
tsc: The TypeScript Compiler - Version 4.5.5
```

Configuring the Compiler.

By default the TypeScript compiler will print a help message when run in an empty project.

The compiler can be configured using a `tsconfig.json` file.

You can have TypeScript create `tsconfig.json` with the recommended settings with:

```
npx tsc --init
```

Which should give you an output similar to:

```
Created a new tsconfig.json with:
TS
  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig.json
```

Here is an example of more things you could add to the `tsconfig.json` file:

```
{
  "include": ["src"],
  "compilerOptions": {
    "outDir": "./build"
  }
}
```

TypeScript simple types.

TypeScript supports some simple types (primitives) you may know.

There are three main primitives in JavaScript and TypeScript.

- `boolean` - true or false values
- `number` - whole numbers and floating point values
- `string` - text values like "TypeScript Rocks"

There are also 2 less common primitives used in later versions of Javascript and TypeScript.

- `bigint` - whole numbers and floating point values, but allows larger negative and positive numbers than the `number` type.
- `symbol` are used to create a globally unique identifier.

Explicit Type

Explicit - writing out the type:

```
let firstName: string = "Dylan";
```

TypeScript Special Types.

Type: any

`any` is a type that disables type checking and effectively allows all types to be used.

The example below does not use `any` and will throw an error:

Example without `any`

```
let u = true;
u = "string"; // Error: Type 'string' is not assignable to type 'boolean'.
Math.round(u); // Error: Argument of type 'boolean' is not assignable to parameter of type 'number
```

Type: unknown

`unknown` is a similar, but safer alternative to `any`.

TypeScript will prevent `unknown` types from being used, as shown in the below example:

```
let w: unknown = 1;
w = "string"; // no error
w = {
  runANonExistentMethod: () => {
    console.log("I think therefore I am");
  }
} as { runANonExistentMethod: () => void}
// How can we avoid the error for the code commented out below when we don't know the type?
// w.runANonExistentMethod(); // Error: Object is of type 'unknown'.
if(typeof w === 'object' && w !== null) {
  (w as { runANonExistentMethod: Function }).runANonExistentMethod();
}
// Although we have to cast multiple times we can do a check in the if to secure our type and have a safer casting
```

TypeScript Special Types.

TypeScript has a specific syntax for typing arrays.

Read more about arrays in our JavaScript Array chapter.

Example

```
const names: string[] = [];
names.push("Dylan"); // no error
```

Typed Arrays

A **tuple** is a typed array with a pre-defined length and types for each index.

Tuples are great because they allow each element in the array to be a known type of value.

To define a tuple, specify the type of each element in the array:

Example

```
// define our tuple
let ourTuple: [number, boolean, string];

// initialize correctly
ourTuple = [5, false, 'Coding God was here'];
```

Named Tuples

Named tuples allow us to provide context for our values at each index.

Example

```
const graph: [x: number, y: number] = [55.2, 41.3];
```

TypeScript Object Types.

TypeScript has a specific syntax for typing objects.

Read more about objects in our JavaScript Objects chapter.

Example

```
const car: { type: string, model: string, year: number } = {
  type: "Toyota",
  model: "Corolla",
  year: 2009
};
```

Try it Yourself »

TypeScript Enums.

An **enum** is a special "class" that represents a group of constants (unchangeable variables).

Enums come in two flavors `string` and `numeric`. Lets start with numeric.

Numeric Enums - Default

By default, enums will initialize the first value to `0` and add 1 to each additional value:

Example

```
enum CardinalDirections {
  North,
  East,
  South,
  West
}
let currentDirection = CardinalDirections.North;
// logs 0
console.log(currentDirection);
// throws error as 'North' is not a valid enum
currentDirection = 'North'; // Error: "North" is not assignable to type 'CardinalDirections'.
```

TypeScript Aliases.

Type Aliases

Type Aliases allow defining types with a custom name (an Alias).

Type Aliases can be used for primitives like `string` or more complex types such as `objects` and `arrays`:

Example

```
type CarYear = number
type CarType = string
type CarModel = string
type Car = {
  year: CarYear,
  type: CarType,
  model: CarModel
}

const carYear: CarYear = 2001
const carType: CarType = "Toyota"
const carModel: CarModel = "Corolla"
const car: Car = {
  year: carYear,
  type: carType,
  model: carModel
};
```

TypeScript Union Types.

Union types are used when a value can be more than a single type.

Such as when a property would be `string` or `number`.

Union | (OR)

Using the `|` we are saying our parameter is a `string` or `number`:

Example

```
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code}.`)
}
printStatusCode(404);
printStatusCode('404');
```

TypeScript Functions.

TypeScript has a specific syntax for typing function parameters and return values.

Read more about functions here.

Return Type

The type of the value returned by the function can be explicitly defined.

Example

```
// the `: number` here specifies that this function returns a number
function getTime(): number {
  return new Date().getTime();
}
```

Parameters

Function parameters are typed with a similar syntax as variable declarations.

Example

```
function multiply(a: number, b: number) {
  return a * b;
}
```

TypeScript Casting.

Casting is the process of overriding a type.

Casting with `as`

A straightforward way to cast a variable is using the `as` keyword, which will directly change the type of the given variable.

Example

```
let x: unknown = 'hello';
console.log((x as string).length);
```

Casting with `<>`

Using <> works the same as casting with `as`.

Example

```
let x: unknown = 'hello';
console.log((<string>x).length);
```

TypeScript Classes.

Members: Types

The members of a class (properties & methods) are typed using type annotations, similar to variables.

Example

```
class Person {
  name: string;
}

const person = new Person();
person.name = "Jane";
```

Members: Visibility

Class members also be given special modifiers which affect visibility.

There are three main visibility modifiers in TypeScript.

- `public` - (default) allows access to the class member from anywhere
- `private` - only allows access to the class member from within the class
- `protected` - allows access to the class member from itself and any classes that inherit it, which is covered in the inheritance section below

Example

```
class Person {
  private name: string;

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

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

const person = new Person("Jane");
console.log(person.getName()); // person.name isn't accessible from outside the class since it's private
```

TypeScript Classes.

Parameter Properties

TypeScript provides a convenient way to define class members in the constructor, by adding a visibility modifiers to the parameter.

Example

```
class Person {
  // name is a private member variable
  public constructor(private name: string) {}

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

const person = new Person("Jane");
console.log(person.getName());
```

Readonly

Similar to arrays, the `readonly` keyword can prevent class members from being changed.

Example

```
class Person {
  private readonly name: string;

  public constructor(name: string) {
    // name cannot be changed after this initial definition, which has to be either at it's declaration or in the constructor.
    this.name = name;
  }

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

const person = new Person("Jane");
console.log(person.getName());
```

TypeScript Interfaces.

Inheritance: Implements

Interfaces (covered here) can be used to define the type a class must follow through the `implements` keyword.

Example

```
interface Shape {
  getArea: () => number;
}

class Rectangle implements Shape {
  public constructor(protected readonly width: number, protected readonly height: number) {}

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

TypeScript Inheritance: Extends.

Inheritance: Extends

Classes can extend each other through the `extends` keyword. A class can only extends one other class.

Example

```
interface Shape {
  getArea: () => number;
}

class Rectangle implements Shape {
  public constructor(protected readonly width: number, protected readonly height: number) {}

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

class Square extends Rectangle {
  public constructor(width: number) {
    super(width, width);
  }

  // getArea gets inherited from Rectangle
}
```

TypeScript Inheritance: Override.

When a class extends another class, it can replace the members of the parent class with the same name.

Newer versions of TypeScript allow explicitly marking this with the `override` keyword.

Example

```
interface Shape {
  getArea: () => number;
}

class Rectangle implements Shape {
  // using protected for these members allows access from classes that extend from this class, such as Square
  public constructor(protected readonly width: number, protected readonly height: number) {}

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

  public toString(): string {
    return `Rectangle[width=${this.width}, height=${this.height}]`;
  }
}

class Square extends Rectangle {
  public constructor(width: number) {
    super(width, width);
  }

  // this toString replaces the toString from Rectangle
  public override toString(): string {
    return `Square[width=${this.width}]`;
  }
}
```

TypeScript Inheritance: Abstract Classes.

Abstract Classes

Classes can be written in a way that allows them to be used as a base class for other classes without having to implement all the members. This is done by using the `abstract` keyword. Members that are left unimplemented also use the `abstract` keyword.

Example

```
abstract class Polygon {
  public abstract getArea(): number;

  public toString(): string {
    return `Polygon[area=${this.getArea()}]`;
  }
}

class Rectangle extends Polygon {
  public constructor(protected readonly width: number, protected readonly height: number) {
    super();
  }

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

TypeScript Generics.

Functions

Generics with functions help make more generalized methods which more accurately represent the types used and returned.

Example

```
function createPair<S, T>(v1: S, v2: T): [S, T] {
  return [v1, v2];
}
console.log(createPair<string, number>('hello', 42)); // ['hello', 42]
```

Classes

Generics can be used to create generalized classes, like Map.

Example

```
class NamedValue<T> {
  private _value: T | undefined;

  constructor(private name: string) {}

  public setValue(value: T) {
    this._value = value;
  }

  public getValue(): T | undefined {
    return this._value;
  }

  public toString(): string {
    return `${this.name}: ${this._value}`;
  }
}

let value = new NamedValue<number>('myNumber');
value.setValue(10);
console.log(value.toString()); // myNumber: 10
```

Thank you