TypeScript Tutorial

TypeScript is JavaScript with added syntax for types.

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. **JavaScript will not throw an error for mismatched types.**

Visual Studio Code, have built-in TypeScript support and can show errors as you write code!

Installing the Compiler

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

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

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

```
npx tsc
```

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
```

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

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

You can open the file in an editor to add those options. This will configure the TypeScript compiler to transpile TypeScript files located in the src/ directory of your project, into JavaScript files in the build/ directory.

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"

When creating a variable, there are two main ways TypeScript assigns a type:

Explicit: let firstName: string = "Dylan";

Implicit: let firstName = "Dylan";

Type: **any**

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

```
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'.
```

Setting any to the special type any disables type checking:

```
let v: any = true;

v = "string"; // no error as it can be "any" type

Math.round(v); // no error as it can be "any" type
```

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
```

The **readonly** keyword can prevent arrays from being changed.

```
const names: readonly string[] = ["Dylan"];

names.push("Jack"); // Error: Property 'push' does not exist on type 'readonly string[]'.

// try removing the readonly modifier and see if it works?
```

Typed Arrays

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

```
// define our tuple

let ourTuple: [number, boolean, string];

// initialize correctly

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

Example

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

Since tuples are arrays we can also destructure them.

Example

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

const [x, y] = graph;
```

Union types

Are used when a value can be more than a single type. Such as when a property would be string or number. 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');
```

Functions

// the **`: number`** here specifies that this function returns a number

```
function getTime(): number {
  return new Date().getTime();
}
```

The type **void** can be used to indicate a function doesn't return any value.

```
function printHello(): void {
  console.log('Hello!');
}
```

Parameters

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

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

Optional Parameters

By default TypeScript will assume all parameters are required, but they can be explicitly marked as optional.

```
// the `?` operator here marks parameter `c` as optional
function add(a: number, b: number, c?: number) {
  return a + b + (c || 0);
}
```

Default Parameters

For parameters with default values, the default value goes after the type annotation:

```
function pow(value: number, exponent: number = 10) {
  return value ** exponent;
}
```

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.

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

Casting with <>

Using <> works the same as casting with as.

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

Classes

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

```
class Person {
  name: string;
}
const person = new Person();
person.name = "Jane";
```

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

Inheritance: Implements

```
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;
  }
}
```

Inheritance: **Extends**

```
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
}
```

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.

```
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}]`;

  }
```

```
}
```

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.

```
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;
  }
}
```