03.03.2023

TypeScript 5

Mikhail Stepanov

Class member access visibility

- public
- protected
- private

2

Other property modifiers

- static
- readonly

public

Default visibility of class members

```
class Employee {
  public code: number;
  public name: string;
  public constructor(code: number, name: string) {
    this.code = code;
    this.name = name;
  }
}


let emp = new Employee(123, "Jake");
emp.code = 321; // OK
emp.name = "Mike"; // OK
```

protected

Visible only to derived classes and declaring class

```
class SalesEmployee extends Employee{
    private department: string;

    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.department = department;
    }
}

let emp = new SalesEmployee("John Smith", 123, "Sales");
emp.empCode; //Compiler Error
```

private

Visible only to declaring class

```
class Employee {
    private empCode: number;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123; // Compiler Error
emp.empName = "Jake";//OK
```

static

Always accessible, aren’t associated with a particular instance of the class

```
class Circle {
  static pi = 3.14159265358979323846;

  public static calculateArea(radius: number) {
    return this.pi * radius * radius;
  }
}

// no need to create an instance of the class
const area = Circle.calculateArea(4);
```

readonly

Prevents assignments to the field outside of the constructor.

```
class Employee {
  public readonly name: string;
  public constructor(name: string) {
    this.name = name;
  }
}

const emp = new Employee("Jake");
emp.name = "Mike"; // Error: readonly property
```

Abstract classes

Similar to interfaces, but can implement some common functionality.

```
abstract class OutputDevice {
    // common functionality
    protected readonly image: string;
    constructor(image: string) {
        this.image = image;
    }

    // abstract method
    abstract print(): void;
}
```

Abstract classes

Child classes should override abstract methods.

```
class Monitor extends OutputDevice {
    // optional override keyword
    override print(): void {
        this.displayImageOnScreen(this.image);
    }
}

class Printer extends OutputDevice {
    override print() {
        const processedImage = this.processImage(this.image);
        this.sendImageToPrint(processedImage)
    }
}
```

Absract class challenge

CodeSandbox

Decorators

A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter.

```
// accepts class, returns modified class
const addProductionYear = (target: { new(): {} }) => class extends target {
  productionYear = new Date().getFullYear();
}

@addProductionYear
class Car {
  ...
}
```

Decorator factories

A Decorator Factory is simply a function that returns the expression that will be called by the decorator at runtime.

```
const setYear = (value: number) => (target: Object, propertyKey: string) => {
  Object.defineProperty(target, propertyKey, { value });
};

class {
  @setYear(1950) // property decorator factory
  public productionYear!: number;
}
```

Method decorators, parameter decorators

A Method Decorator can be used to observe, modify, or replace a method definition.
A Parameter Decorator is applied to the function for a class constructor or method declaration

```
class LoggableCar implements WithProductionYear {
  @log
  setYear(@loggable year: number, @loggable model: string) {
    this.productionYear = year;
    this.model = model;
  }
}
```

Namespaces

Namespaces are an outdated way to organize TypeScript code. ES2015 module syntax is now preferred (import / export).

```
// if namespace is in other file, you should reference it with triple slash syntax
/// <reference path = "SomeFileName.ts" />
namespace N {
    // We can also make nested namespaces
    export namespace M {
        export const Surname = "S";
    }
}
```

.d.ts

d.ts files called declaration files.
Mostly used for describing types and interfaces of existing JavaScript libraries.

```
// lib.es5.d.ts
interface Math {
    ...
    /**
     * Returns the smallest integer greater than or equal to its numeric argument.
     * @param x A numeric expression.
     */
    ceil(x: number): number;
    ...
}
```

TS config

tsconfig.json is a recommended way to set up compilation configuration. The presence of a tsconfig.json` file in a directory indicates that the directory is the root of a TypeScript project.

TS config base

You may want to reuse a base configuration file

```
{
  "extends": "@tsconfig/node12/tsconfig.json",
  ...
```

TS config include and exclude properties

Include and exclude patterns for your project' files

```
  ...
  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

TS config target and lib

The target setting changes which JS features are downleveled and which are left intact

Lib provides a set of type definitions for built-in JS APIs

```
{
  "target": "ES5", // target ES5 browsers
  "lib": "ES6",    // but use ES6 polyfills like Promise
  ...
```

Strict null checks

When strictNullChecks is false, null and undefined are effectively ignored by the language

```
// @strictNullChecks

function getUser(): User | undefined {
    return getUserFromDatabase();
}

// ERROR with strictNullChecks enabled, could be undefined
const user: User = getUser();
```

Questions

Evolution Engineering | ACCELERATING EVOLUTION