

Technologies and Web Programming

Angular Framework



Angular Framework

TypeScript Language

TypeScript Language



- TypeScript starts from the same syntax and semantics of JavaScript.
 - It can use existing JavaScript code, incorporate popular JavaScript libraries, and its compiled code can be called from JavaScript.
- TypeScript compiles to clean and simple JavaScript code which runs on:
 - any browser;
 - Node.js;
 - any JavaScript engine supporting ECMAScript 3+.

TypeScript Language



- TypeScript greatest value it's all about defining and using Types for JavaScript. Even, if they are of optional use.
- It enables JavaScript practices like:
 - static checking;
 - and code refactoring.
- Types let you define interfaces between software components and give you more control on the behavior of JavaScript libraries.
- Also, TypeScript provides type inference, for code static verification.

Variables Declaration



- `var x = 1; // deprecated because its flaws`
- `let y = 2; // new and strong declaration`
- `const z = 3; // constants`
- `let arr = [10, 20];`
- `let [a, b] = arr; // array destructuring`
- `Let o = { a: 1, b: 'hello', c: 10 }`
- `Let {a, c} = o; // object destructuring`

Basic Types (i)



- Basic types for simple data units:
 - boolean
 - `let isfull: boolean = true;`
 - number
 - `let dec: number = 10.5; // decimal`
 - `let hex: number = 0xFF; // hexadecimal`
 - `let oct: number = 0o373 // octal`
 - `let bin: number = 0b1010 // binary`
 - string
 - `let name: string = "John";`
 - `name = 'Jones';`
 - `let s: string = `Your name is: ${name}`;`

Basic Types (ii)



- array
 - `let list: number[] = [10, 20, 30];`
 - `let list: Array<number> = [10, 20, 30];`
- tuple
 - `let t: [Boolean, number];`
 - `t = [false, 100]; // ok`
 - `t = [100, false]; // error`
- enum
 - `enum Color {red, green, blue}`
 - `let c: Color = Color.green;`

Basic Types (iii)



- any
 - let what: any = 100;
 - what = true; // ok
- void (nothing, the opposite of any)
 - let nothing: void = null; // or undefined
 - normally used for return type of a function
- null and undefined
 - null is the absence of a value in a variable
 - undefined is the absence of definition of a variable

Basic Types (iv)



- `never`
 - type of values that never occur
 - usually used as function type for never ending functions
- type assertion (cast like)
 - `let avalue: any = "this is a string value";`
 - `let slength: number = (<string>avalue).length;`

Functions (i)



- Like in JavaScript, TypeScript functions can be created both as named functions or as anonymous functions and typed.
- **Examples:**

```
function add(x: number, y: number): number {  
    return x + y;  
}  
let sum = add(1, 2);
```

- **or:**

```
let sum = function(x: number, y: number): number {  
    return x + y;  
};
```

Functions (ii)



- Defining function types:
 - Function typing includes two parts: parameters and return type.
 - In the next example, the function to be used must have two parameters of type number and a return type of type number.

```
let sum: (a: number, b: number) => number =  
  function(x: number, y: number): number {  
    return x + y;  
  };
```

Functions (iii)



- **Default parameters:**

```
function myname(fname: string, lname: 'Burton') {  
    return fname + " " + lname;  
}
```

- **Optional parameters:**

```
function myname(fname: string, lname?: string) {  
    if (lname)  
        return fname + " " + lname;  
    else  
        return fname;  
}
```

Functions (iv)



- Fat Arrow functions
 - It's a way to write functions in a shorthand notation.
 - Examples:

```
let sum = (a, b) => a + b;  
s = sum(10, 20);
```

- Or

```
let data = [1, 2, 3];  
let s = 0;  
data.forEach((x) => s += x);
```

Interfaces (i)



- One of TypeScript's core principles is that type-checking focuses on the shape that values have.
- They are a powerful way of defining contracts within your code as well as contracts with code from outside.

Interfaces (ii)



- **Example:**

```
interface Hello {  
    mesg: string;  
    name: string;  
}  
  
function printHello(obj: Hello)  
{  
    console.log(obj.mesg + " " + obj.name + "!!!");  
}  
  
let myObj = {idobj: 10, mesg: "Hello myfriend", name: "John",  
             fullname: "John Simons"};  
printHello(myObj);
```

Classes (i)



- TypeScript allow developers to use object-oriented techniques and approaches, based on classes, to program their web scripts.
- It compiles down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes (ii)



- **Example:**

```
class Welcome {  
    name: string;  
    constructor(name: string) {  
        this.name = name;  
    }  
    say() {  
        return `Hello ${this.name}! You're Welcome!!!`;  
    }  
}
```

```
let w = new Welcome("Robert");  
console.log(w.say());
```

Classes - Inheritance



- **Example:**

```
class Animal {  
    move(distance: number = 0) {  
        return `Animal moved ${distance} meters.`;  
    }  
}  
  
class Dog extends Animal {  
    bark() {  
        return 'Woof! Woof!';  
    }  
}  
  
let dog = new Dog();  
let ss = dog.bark();  
ss += "\n" + dog.move(10);  
ss += "\n" + dog.bark();  
console.log(ss);
```

Classes – Overloading (i)



- Example:

```
class Animal {  
    name: string;  
    constructor(n: string) {  
        this.name = n;  
    }  
    talk() {  
        return this.name + " is talking: ";  
    }  
}
```

Classes – Overloading (ii)



- **Example:**

```
class Dog extends Animal {  
    constructor(name: string) {  
        super(name);  
    }  
    talk() {  
        return super.talk() + 'Woof! Woof!';  
    }  
}
```

```
let dog = new Dog('Ben');  
let ss = dog.talk();  
console.log(ss);
```

Modules (i)



- Modules are executed within their own scope, not in the global scope.
- Variables, functions, classes, etc. declared in a module are not visible outside the module.
- Modules are declarative – the relationships between modules are specified in terms of imports and exports at the file level.
- In TypeScript, any file containing a top-level import or export is considered a module.
- Conversely, a file without any top-level import or export declarations is treated as a script whose contents are available in the global scope.

Modules (ii)



- **Example:**

```
// File: Validation.ts
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

```
// File: ZipCodeValidator.ts
import { StringValidator } from "../Validation";
export const numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Book



- A book where to learn all about TypeScript language:
 - The TypeScript Handbook
 - <https://www.typescriptlang.org/docs/handbook/intro.html>