TypeScript

Junghoo Cho

cho@cs.ucla.edu

TypeScript - Junghoo Cho - cho@cs.ucla.edu

TypeScript

- *Superset* of JavaScript (a.k.a. JavaScript++)
  - New features: types, interfaces, decorators, …
  - All additional TypeScript features are *strictly optional* and are not rec
  - Any JavaScript code is also a TypeScript code!

Example

- Types can be added to functions and variables

```
function hello(greeting: string): string
{
    return "Hello, " + greeting + "!";
}

let world: string = "world";
console.log(hello(world));
```

  - The origin of the name: JavaScript with Type!

Transpilation

- TypeScript code is “compiled” to JavaScript code using TypeS compiler

```
$ npm install -g typescript
$ tsc hello.ts    // compiles `hello.ts` to `hello.js`
$ node hello.js   // runs compiled JavaScript code
```

- TypeScript Playground

Static Type Checking

```
function hello(greeting: string): string
{
    return "Hello, " + greeting + "!";
}

hello([0, 1, 2]);
```

- Compiler produces an error due to type mismatch

```
$ tsc hello.ts
hello.ts(6,11): error TS2345: Argument of type 'number[]'...
```

Why Static Type Checking?

- Q: Why would anyone want this?
- A: Static type checking can potentially make large-scale cod
  manage
  - Compile-time error vs run-time error
  - Rigidity vs flexibility

Type Annotations

```
let a: number = 1;
let b: string = "A";
let c: boolean = true;
let d: number[] = [1, 2];                          // array
let e: [number, string] = [1, "A"];                // tuple
let f: (number | string) = 1; // or "A"            // union
let g: any = "A";  // or 1, [1, 2], ...           // any
let h: void = undefined; // or null                // void
let i: never; // nothing can be assigned           // never

let j: {x: number, y: string} = {x:1, y:"A"};      // object

function hello(name: string): void {               // function
    console.log(`Hello, ${name}!`);
}
let k: (x: string) => void = hello;                // function
```

Type Compatibility: Basic Types

- Three basic type values cannot be assigned to a different typ

```
// all errors!
let a: number = true;
let b: string = true;
let c: boolean = 1;
let d: string = 1;
let e: number = "1";
let f: boolean = "1";
```

- `undefined` and `null` can be assigned to any types
  - But not the other way around, of course
  - `--strictNullChecks` disallows `undefined` or `null` for other types

Type Compatibility: Object Types

- In assignment, object of type `A` can be assigned to a variable if their structure is "compatible"
  - `let b: B = new A();`
  - `A`'s properties should be a superset of `B`'s

Type Compatibility: Object Types

- Example

```
class Point2D {
    x: number = 0;
    y: number = 0;
};
function plot(p: Point2D): void {
    console.log("("+ p.x + ", " + p.y +")");
}
let point_3d = { x: 1, y: 2, z: 3 };
plot(point_3d);
```

- No error because `point_3d` is compatible with `Point2D`

Type Conversion

- Converting primitive types: `String()`, `Number()`, `Boolean()`

```
let a: number = Number("1");
let b: string = String(2);
let c: boolean = Boolean("true");
```

- Object type conversion: `as` and `<>` operators

```
let input = document.querySelector('input[type="text"]') as HT
let input = <HTMLInputElement>document.querySelector('input[ty
```

  - Two operators are equivalent
  - `querySelector()` returns `HTMLElement` type
    - `HTMLInputElement` is a subclass of `HTMLElement`

Enum Type

- Like Java/C++ enum

```
enum myEnumType { A, B, C };
let x: myEnumType = myEnumType.A;
console.log(x);
```

Functions (1)

- In JavaSciprt, missing parameters are OK and are bound to u

```
function sum(a, b, c)
{
    return (a || 0) + (b || 0) + (c || 0);
}
console.log(sum(10)); // OK!
```

- In TypeScript, all function parameters must be passed

```
function sum(a: number, b: number, c: number): number
{
    return (a || 0) + (b || 0) + (c || 0);
}
console.log(sum(10)); // error!
```

Functions (2)

- *Optional parameter*: suffix `?` indicates optional parameter

```
function sum(a: number, b?: number, c?: number): number
{
    return (a || 0) + (b || 0) + (c || 0);
}
console.log(sum(10)); // OK!
```

- But `sum(a?, b, c?)` is not allowed, of course

Function: Rest Parameters

- Like in JavaScript, rest operator can be used to pass variable parameters (ECMAScript 2015)

```
function f(a, b, ...c) {
    console.log("a = " + a);
    console.log("b = " + b);
    console.log("c = " + c);
}
f(1, 2, 3, 4);
```

Classes

```
// JavaScript
class Point {
    constructor(x, y)
    {
        this.x = x;
        this.y = y;
    }
}
```

```
// TypeScript
class Point {
    private x: number;
    public  y: number;
    constructor (x: numbe
                 y: numbe
    {
        this.x = x;
        this.y = y;
    }
}
```

- Explicit member property declaration is required
  - `constructor` required if no initial value is set in declaration
- Access modifiers: `public`, `private`, `protected`
  - `public` by default. `#property` is also private

Class Constructor Shorthand

- Adding access modifier to constructor parameter automatic
  the parameter as a property

```
class Point {
    private x: number;
    public  y: number;
    constructor (x: number,
                 y: number)
    {
        this.x = x;
        this.y = y;
    }
}
```

```
class Point {
    constructor (private
                 public
    {}
}
```

- Above two are equivalent

Non-Null Assertion Operator

- ES2020 introduced optional chaining operator

```
console.log(obj?.name) // returns undefined if obj is undefine
```

- In addition, TypeScript has *non-null assertion operator*, `!`
  - Tell the compiler that the value is never `null` or `undefined`
  - Compiler can use this info for better type inferencing and optimizat

```
console.log(obj!.name)
```

Interfaces (1)

- Like Java interface

```
interface Person {
    firstName: string;
    middleName?: string;
    lastName: string;
}
function hello(p: Person): void {
    console.log("Hello, " + p.firstName);
}
hello({firstName: "James", lastName: "Dean"});
```

Interfaces (2)

- Can be used to define a “function type” as well

```
interface SearchFunc {
    (str: string, pattern: string): boolean;
}
// same as "type SearchFunc = (str: string, pattern: string) =

let mySearch: SearchFunc;
mySearch = function (source: string, subString: string): boole
    let result = source.search(subString);
    return result > -1;
};
```

Generics

- Like Java generics: "Type-parameterized" class/function

```
// Generic class
class Dot<T> {
    public x: T;
    constructor(x: T) { this.x = x; }
}
let s = new Dot<number>(1);

// Generic function
function log<T>(arg: T): void
{  console.log(arg);  }
log<number>(1);
```

- In TypeScript, `Promise` can be generic type, like `Promise<st`
  - Q: What is `string` type for?

Decorators

- Syntax: `@decorator_name`
  - Decorator can be added to certain declarations (class, method, …)
  - Decorator can modify various aspects of declared entities
- Example:

```
@sealed     // <- decorator
class Greeter {
    constructor(public greeting: string) {}
    ...
}
```

  - "Seal" `Greeter` objects
    - Sealed object: property values may change but structure is fixed
    - e.g., no new property or method can be added

Getter and Setter (ECMAScript 2015)

```
class Dot {
    _x = 0;
    set x(v) { this._x = v;  }                // setter
    get norm() { return Math.abs(this._x); } // getter
};

let p = new Dot();
console.log("p.norm = " + p.norm);       // no parenthesis!
console.log("Before: p._x = " + p._x);
p.x = 20;                                // not a function call!
console.log("After:  p._x = " + p._x);
```

What We Learned

- Key TypeScript features
  - Type annotator
  - Type compatibility, type conversion
  - Enum type, Function type
  - TypeScript class
  - Interface, Generic, Decorator
  - Getter and setter
- We mainly focused on what is frequently used in Angular

References

- TypeScript