TypeScript and Webpack

Unifying the JS Ecosystem

Kimberly Tran

Aug 4th 2016

JS Fragmented

- JavaScript versions
- Poor IDE tooling
- Ad-hoc OOP
- Client-side vs Server-side paradigms
- Overlapping tools
- Ad-hoc modules

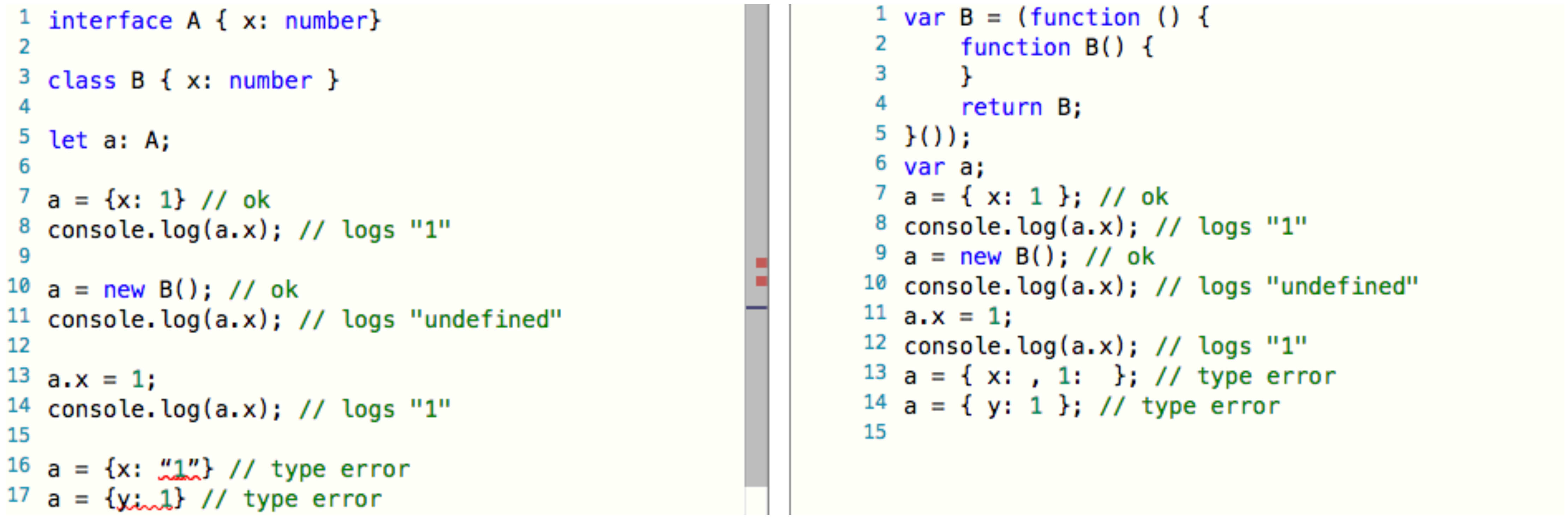
TypeScript = ES6

- ES3 (“JavaScript”), ES5 (use strict), ES6 (classes & modules)
- Write ES6-style code everywhere
  - TypeScript is ES6 with static type checking
- TypeScript will transpile to ES5 (or ES3 or ES6)
- Wide support (IE9) for ES5
- As a developer: ES6 or bust

TypeScript Tooling

- Static type checking
  - Rich type system based on **structural typing**
- Refactoring w/ confidence
- Language service means choice of IDEs
  - Navigation support
  - Auto-completion

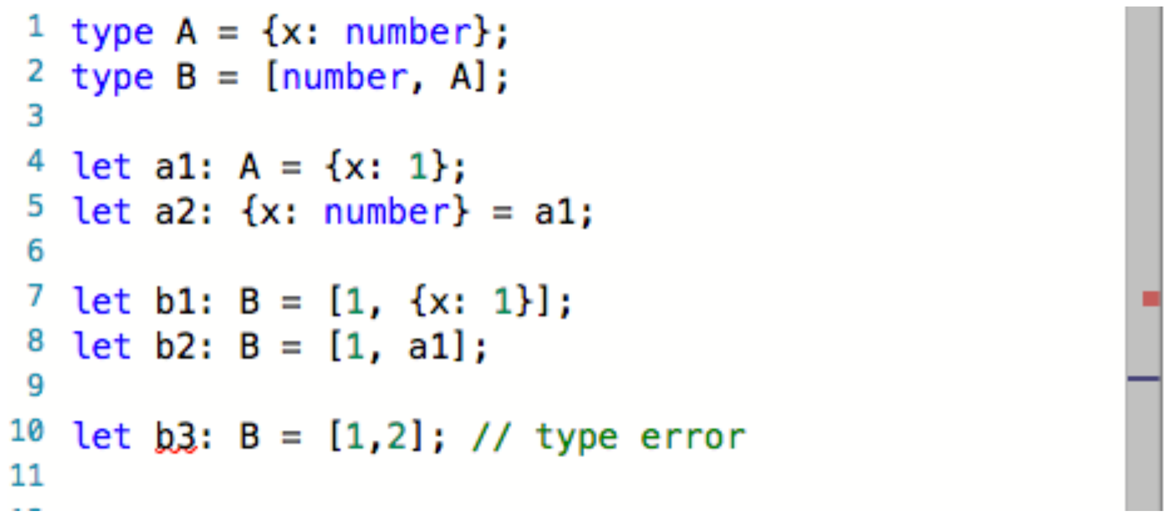
Structural typing



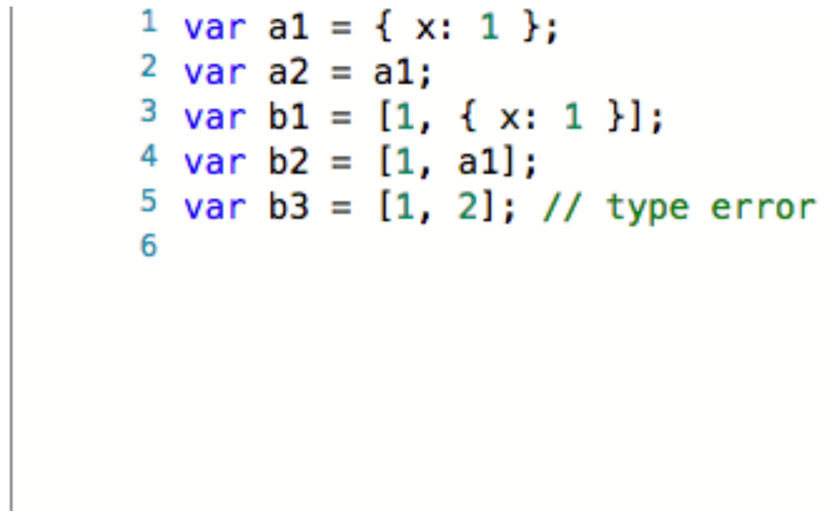
```
interface A { x: number}

class B { x: number }

let a: A;

a = {x: 1} // ok
console.log(a.x); // logs "1"

a = new B(); // ok
console.log(a.x); // logs "undefined"

a.x = 1;
console.log(a.x); // logs "1"

a = {x: "1"} // type error
a = {y: 1} // type error
```

```
var B = (function () {
    function B() {
    }
    return B;
}());
var a;
a = { x: 1 }; // ok
console.log(a.x); // logs "1"
a = new B(); // ok
console.log(a.x); // logs "undefined"
a.x = 1;
console.log(a.x); // logs "1"
a = { x: , 1:  }; // type error
a = { y: 1 }; // type error
```

Structural typing

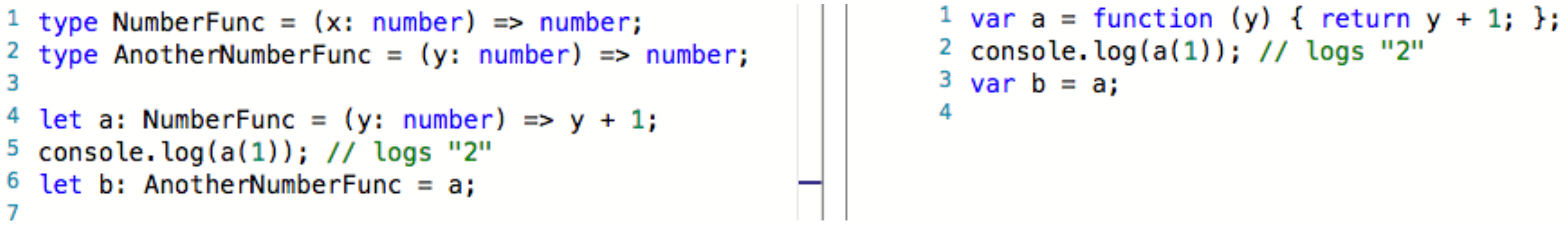


```
type A = {x: number};
type B = [number, A];

let a1: A = {x: 1};
let a2: {x: number} = a1;

let b1: B = [1, {x: 1}];
let b2: B = [1, a1];

let b3: B = [1,2]; // type error

```



```
var a1 = { x: 1 };
var a2 = a1;
var b1 = [1, { x: 1 }];
var b2 = [1, a1];
var b3 = [1, 2]; // type error

```

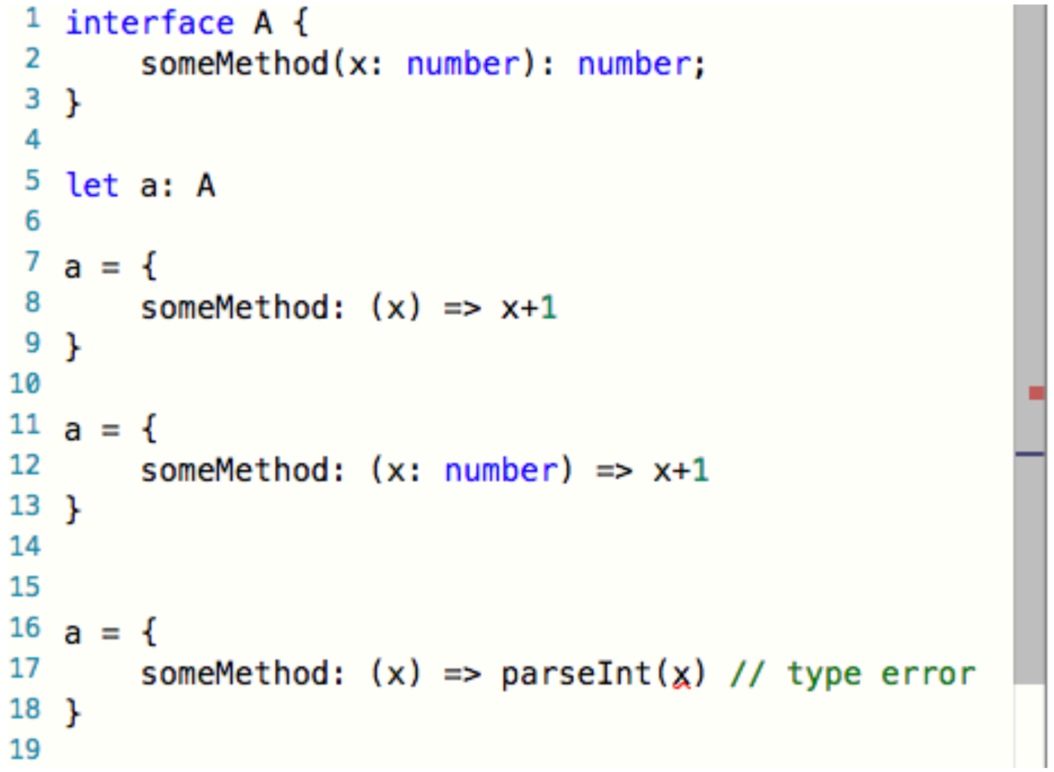
Structural typing



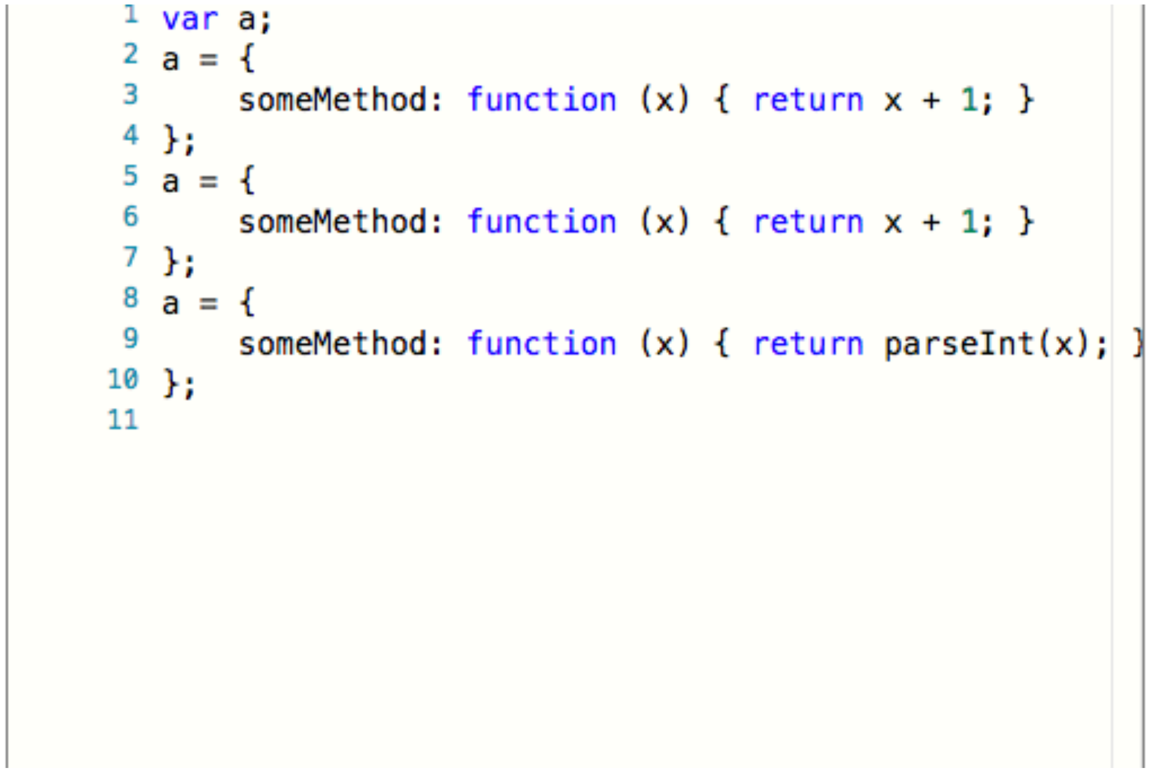
```
type NumberFunc = (x: number) => number;
type AnotherNumberFunc = (y: number) => number;

let a: NumberFunc = (y: number) => y + 1;
console.log(a(1)); // logs "2"
let b: AnotherNumberFunc = a;
```

```
var a = function (y) { return y + 1; };
console.log(a(1)); // logs "2"
var b = a;
```

Type inference

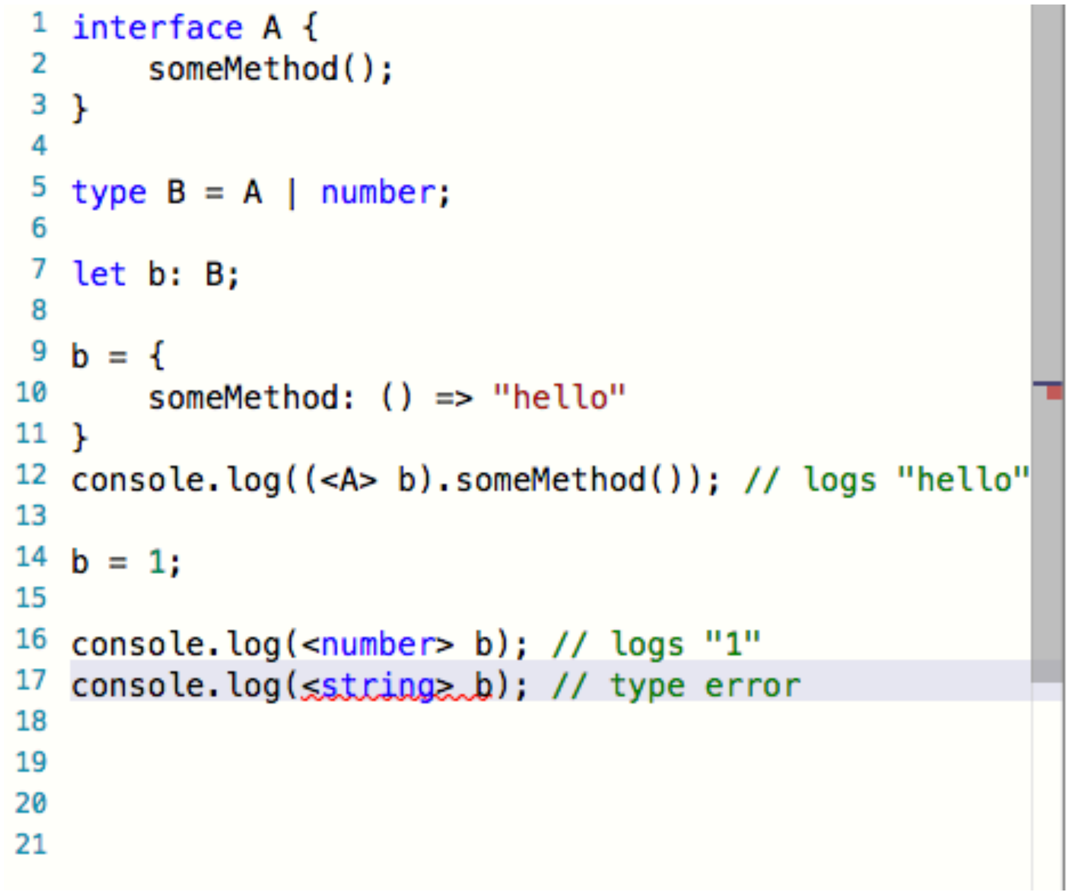


```
interface A {
    someMethod(x: number): number;
}

let a: A

a = {
    someMethod: (x) => x+1
}

a = {
    someMethod: (x: number) => x+1
}


a = {
    someMethod: (x) => parseInt(x) // type error
}

```

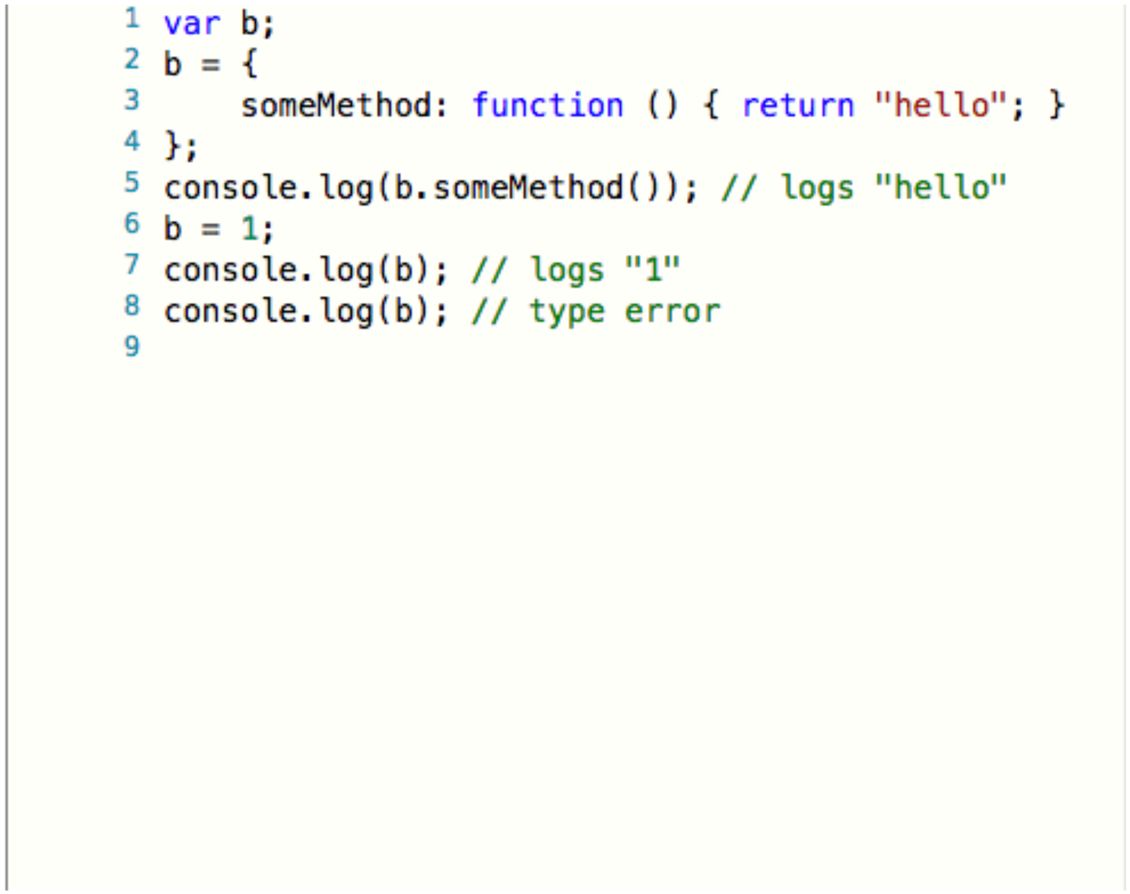


```
var a;
a = {
    someMethod: function (x) { return x + 1; }
};
a = {
    someMethod: function (x) { return x + 1; }
};
a = {
    someMethod: function (x) { return parseInt(x); }
};

```

Union types

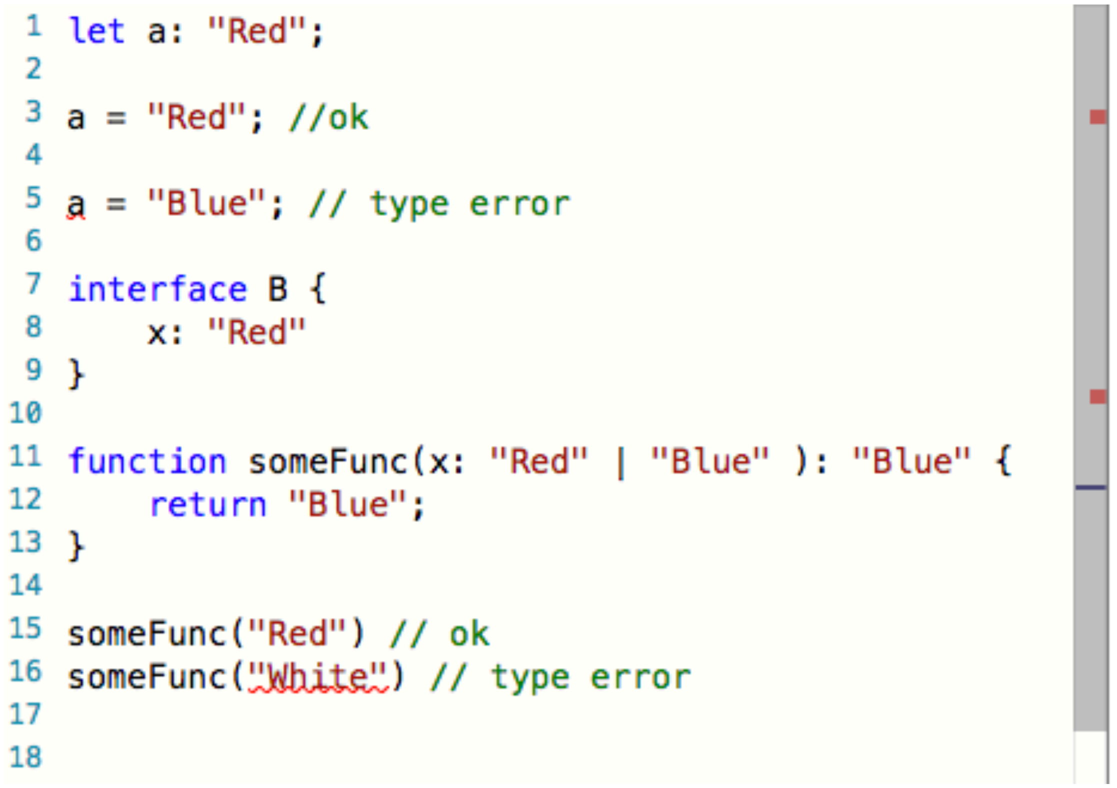


```typescript
interface A {
    someMethod();
}

type B = A | number;

let b: B;

b = {
    someMethod: () => "hello"
}
console.log((<A> b).someMethod()); // logs "hello"

b = 1;

console.log(<number> b); // logs "1"
console.log(<string> b); // type error
```

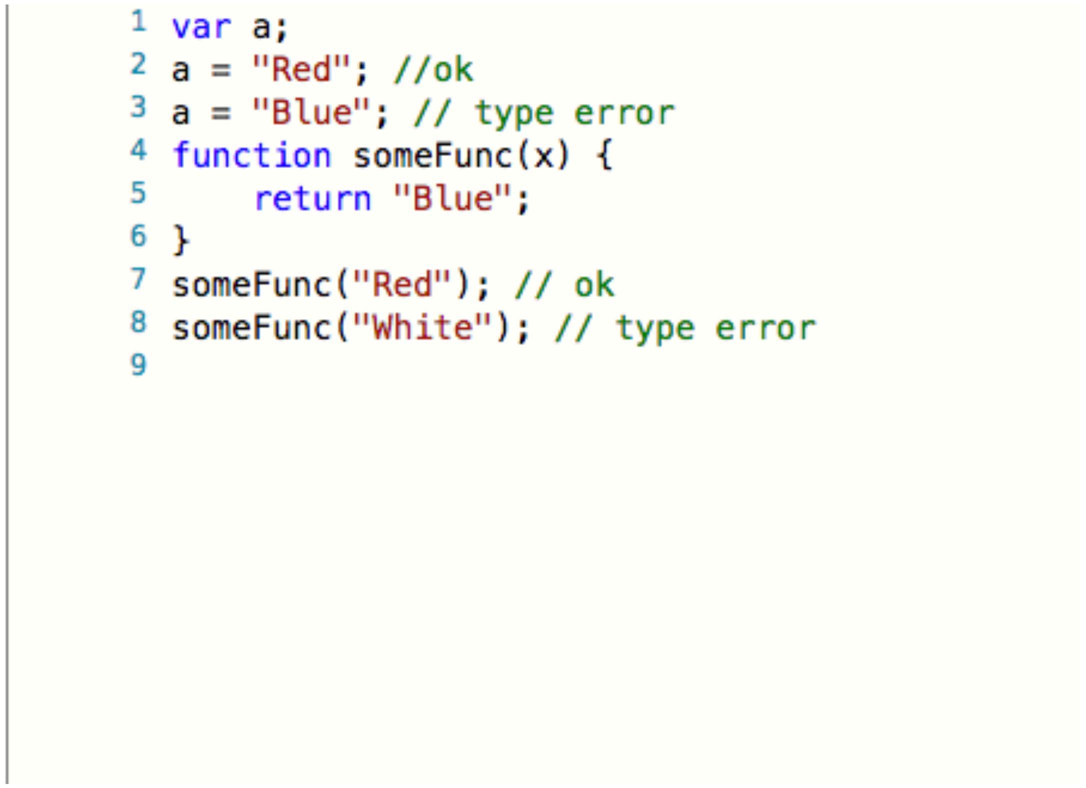


```javascript
var b;
b = {
    someMethod: function () { return "hello"; }
};
console.log(b.someMethod()); // logs "hello"
b = 1;
console.log(b); // logs "1"
console.log(b); // type error
```

String Literal Types



```
let a: "Red";

a = "Red"; //ok

a = "Blue"; // type error

interface B {
    x: "Red"
}

function someFunc(x: "Red" | "Blue" ): "Blue" {
    return "Blue";
}

someFunc("Red") // ok
someFunc("White") // type error


```

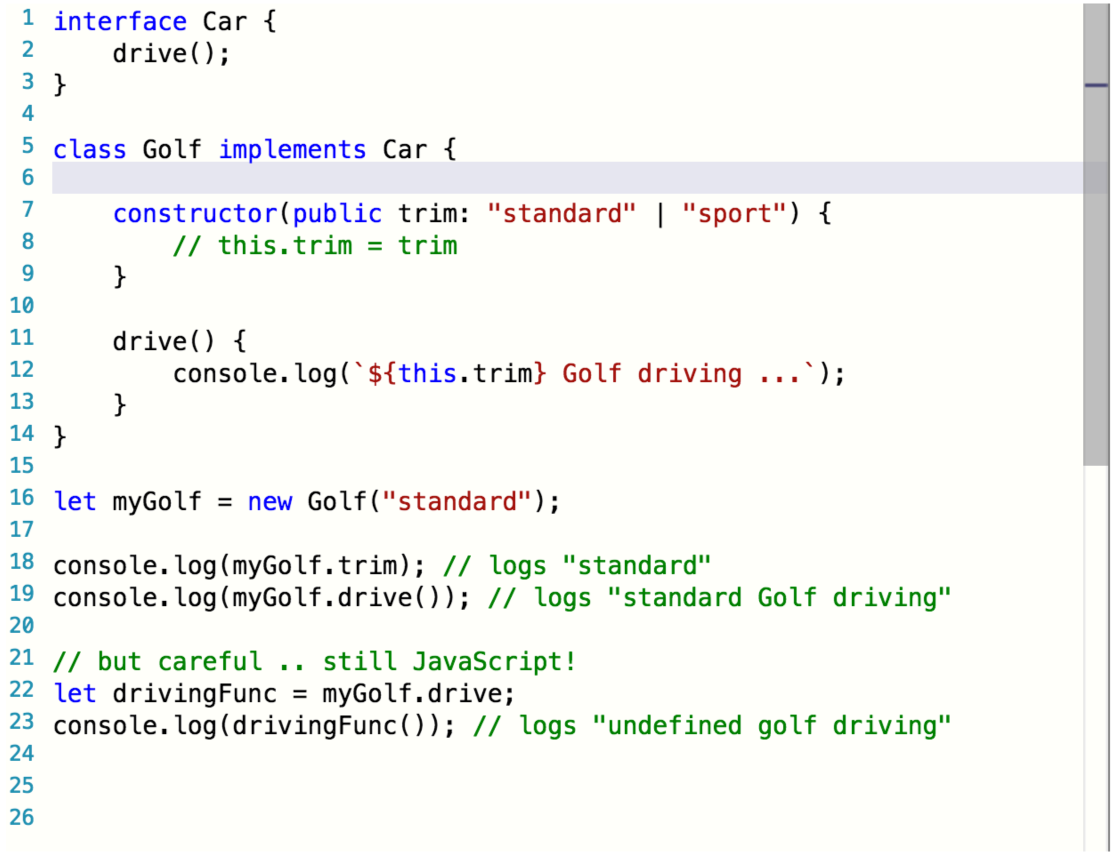


```
var a;
a = "Red"; //ok
a = "Blue"; // type error
function someFunc(x) {
    return "Blue";
}
someFunc("Red"); // ok
someFunc("White"); // type error

```

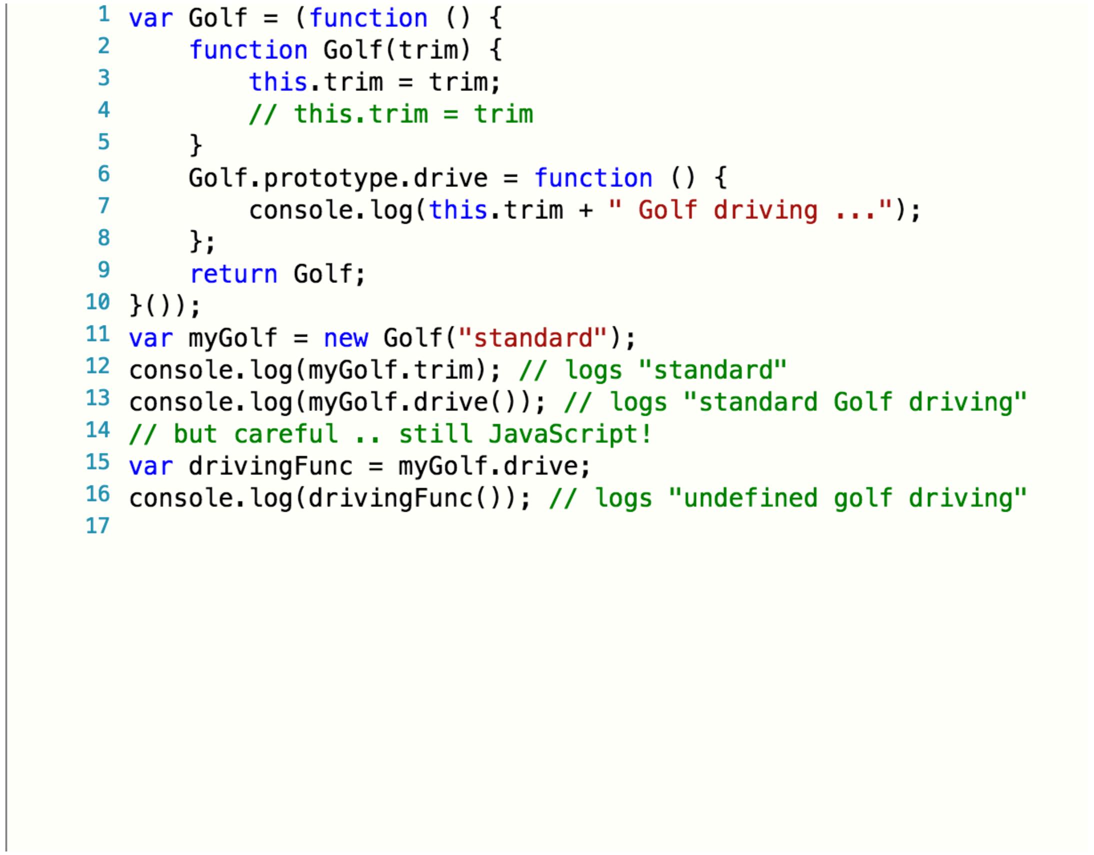
Familiar OOP

- Classical inhertance
- Classes and Interfaces are part of the language
    - Finally, one way to instantiate a class
- Defined in terms of ES5 semantics
    - Complete inter-op with ES5

Classical Inheritance



```typescript
interface Car {
    drive();
}

class Golf implements Car {

    constructor(public trim: "standard" | "sport") {
        // this.trim = trim
    }

    drive() {
        console.log(`${this.trim} Golf driving ...`);
    }
}

let myGolf = new Golf("standard");

console.log(myGolf.trim); // logs "standard"
console.log(myGolf.drive()); // logs "standard Golf driving"

// but careful .. still JavaScript!
let drivingFunc = myGolf.drive;
console.log(drivingFunc()); // logs "undefined golf driving"
```



```javascript
var Golf = (function () {
    function Golf(trim) {
        this.trim = trim;
        // this.trim = trim
    }
    Golf.prototype.drive = function () {
        console.log(this.trim + " Golf driving ...");
    };
    return Golf;
}());
var myGolf = new Golf("standard");
console.log(myGolf.trim); // logs "standard"
console.log(myGolf.drive()); // logs "standard Golf driving"
// but careful .. still JavaScript!
var drivingFunc = myGolf.drive;
console.log(drivingFunc()); // logs "undefined golf driving"
```

Demo: Type-safe promises

- Typings are great for exploring APIs
- Example: The Q implementation of Promise/A+
    - https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/q/Q.d.ts

Span Client/Server

- Single Page App
  - JavaScript in the browser
  - C# (or pick your language) on the server
- TypeScript
  - One language
  - Same libraries
  - Share code

Example: JSON REST APIs

- Shared interfaces defines request, response objects
- Exact same files shared
- Server declares JSON in certain shape
- Client declares JSON in the same shape

Encapsulation

- How to share code?
- Server-side: node package manager (npm)
- Client-side modules
  - Historically, none
  - How to prevent naming collisions?
  - Closures
- Runtime loading of modules
  - NodeJS “require(…)”
  - File sysem based

Client-side modules

- How to load modules in the browser?
- Bundlers
    - Browserify
    - Webpack

Client-side development

- Needs
  - Cache-busting
  - Copy assets
  - Annotations for AngularJS DI
  - Compile CSS
  - Load JavaScript from HTML
- ”Transpilation” always needed
- Lots of tools

Client-side asset pipeline

- Task runners
  - Grunt
  - Gulp
- Dependency managers (two!)
  - Npm
  - Bower
- JavaScript processors
  - Babel
- CSS processors
- HTML processors
- Injecting <script> tags into HTML

Transpilation always needed

- TypeScript
  - Fullfills JavaScript processing needs
  - Code ES6, target ES3/ES5/ES6
- Webpack
  - Module loading needs
  - Asset pipeline needs

Webpack for asset pipelining

- Compiles LESS (or SASS) -> CSS
- Compiles HTML templates
- Compiles images
- Compiles fonts
- Into…
  - One (or more if you wish) JavaScript bundles
- Entire SPA loaded with two HTTP requests
  - Index.html
  - Bundle.js

Webpack bundles

- Not really magic, “just” DOM manipulation
- Parses CSS at build times
  - Looks for module imports
  - Builds dependency trees
  - Hand off to ”loaders”
- Loaders
  - A specific loader for CSS
  - A specific loader for TypeScript
- Example: CSS => bundle as JS module
  - Plus some code to extract CSS
  - Insert into DOM

Resources

- TypeScript
  - https://www.typescriptlang.org/play/
- Webpack
  - https://webpack.github.io/docs/tutorials/getting-started/

- Twitter, Github: @kayjtea