TypeScript

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Objectives

- What is TypeScript?
- Installation and Usage
- Language Features

What is TypeScript?

- TypeScript is a superset of JavaScript
  - all valid JavaScript is valid TypeScript
- Includes features from ECMAScript 6 (ES6)
  - classes
  - modules
  - arrow functions
- Adds features unique to TypeScript
  - type annotations
  - interfaces
  - declarations

TypeScript Compilation

- No browser can execute TypeScript
  - must be “transpiled” to JavaScript
- ES6/TypeScript features are transformed or stripped from compiled output
  - ES6 classes rewritten as traditional constructor functions with prototype objects
  - ES6 modules rewritten using CommonJS or AMD conventions
  - TypeScript type annotations disappear (only used during development and not at runtime)

Installation

- TypeScript compiler is written in TypeScript
  - and then compiled to JavaScript…
- Runs on any OS
  - via Node.js
- Integrates with Visual Studio 2013+
  - via official extension from Microsoft or 3rd party extensions like Web Essentials
- Integration with other editors is available
  - Vim, Emacs, Sublime Text

Installation via Node.js

- Node.js and npm must be installed and in your “PATH”
  - visit http://nodejs.org/download/ to download installers for Windows, Mac OS X, or get information on how to install via other package managers or even build from source

```
npm install --global typescript
```

- Once installed, use tsc to compile .ts files to .js files

```
tsc app.ts
```

Integration with Visual Studio

- Does NOT require Node.js to be installed
  - download from http://www.typescriptlang.org/
- Adds “HTML Application with TypeScript” project template
  - adds “BeforeBuild” target to project file
  - can be copied into other project/MSBuild files
  - Build Action for .ts files should be “TypeScriptCompile”
- Inside .ts files, developer experience radically improved
  - syntax highlighting
  - IntelliSense
  - Go To Definition
  - Find All References
  - etc

Usage

- For client-side usage
  - since browsers can't execute TypeScript, HTML must reference compiled .js files and not .ts files
  - will need to compile using AMD module format and include and use require.js (or other AMD loader) in your pages
- For server-side usage (probably Node.js)
  - Node.js will ignore .ts files and load .js files
  - will need to compile using CommonJS module format (default)

Arrow Functions

- Syntactic sugar for function expressions
  - preservers “this” to be the same as outer function

```
var add = (a, b) => a + b;

// var add = function(a, b) { return a + b; }
```

Arrow Functions and “this”

- TypeScript input:

```
var person = {
    name: 'Anders',
    init: () => {
        document.getElementById('speak').onclick =
            e => alert('Hi, my name is ' + this.name + '!');
    }
};
```

- JavaScript output:

```
var person = {
    name: 'Anders',
    init: function () {
        var _this = this;
        document.getElementById('speak').onclick = function (e) {
            return alert('Hi, my name is ' + _this.name + '!');
        };
    }
};
```

Default Argument Values

- TypeScript input:

```
function add(a, b = 0) {
    return a + b;
}
```

- JavaScript output:

```
function add(a, b) {
    if (typeof b === "undefined") { b = 0; }
    return a + b;
}
```

Classes

- ES6 classes are syntactic sugar for constructor functions plus prototypes
  - syntax is reminiscent of Java
  - uses class, extends, super keywords
  - constructors look like functions named “constructor”

Classes

- TypeScript input:

```
class Person {
    name;
    constructor(name) {
        this.name = name
    }
    speak() {
        alert('Hi, my name is ' + this.name + '!');
    }
}
```

Classes

- JavaScript output:

```
var Person = (function () {
    function Person(name) {
        this.name = name;
    }
    Person.prototype.speak = function () {
        alert('Hi, my name is ' + this.name + '!');
    };
    return Person;
})();
```

Data Hiding

- Class members are public by default
  - can be declared private
- TypeScript-aware tools can prevent access to private members via compile-time errors
  - private members still accessible at runtime

```
class Person {
    public name;
    private age;

    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

Constructor Properties

- “public” and “private” keywords can prefix constructor arguments
  - adds properties to class and initializes in constructor

```
// TypeScript Input
class Person {
    constructor(public name, private age) {}
}
```

```
// JavaScript Output
var Person = (function () {
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }
    return Person;
})();
```

Inheritance

- Classes can have a single base class
  - but can implement multiple interfaces
- Base class members are accessible via super reference
  - derived classes with constructors must explicitly call super() to invoke base class constructor

Inheritance

- TypeScript input:

```
class Dog extends Animal {
    constructor(name) {
        super(name);
    }
    speak() {
        super.speak();
        alert('Woof!');
    }
}
```

Inheritance

- JavaScript input:

```
var Dog = (function (_super) {
    __extends(Dog, _super);
    function Dog(name) {
        _super.call(this, name);
    }
    Dog.prototype.speak = function () {
        _super.prototype.speak.call(this);
        alert('Woof!');
    };
    return Dog;
})(Animal);
```

Modules

- TypeScript provides module abstraction supported with syntax
  - compiler translates modules, exports, and imports into appropriate implementation (CommonJS or AMD)

Type Annotations

- Variables, function arguments, function return values, class and interface members can optionally have type annotations attached to them
  - only used by developer tools
  - not included in compiled output
- Types can be specified in three ways
  - predefined type keywords
  - type names (module, class, or interface names)
  - type literals

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

Predefined Type Keywords

- Represent primitive JavaScript types
  - any (default type when nothing else is indicated)
  - bool
  - number
  - string
  - void (only usable for function return types)

Type Literals

- Different forms of type literals
  - object
  - array
  - function
  - constructor

Object Type Literals

- Specify members expected to be in objects
  - each member can have its own signature
- Member signatures
  - call
  - construct
  - index
  - property
  - function
- Object type literals are surrounded by curly braces and appear after ":", never "="

```
var foo: {
    // Object type literal members go here.
};
```

Call Signatures

- Represents a callable object (a function)
  - can be overloaded

```
var add: {
    (a: number, b: number): number;
    (numbers: number[]): number;
};

add = function(a, b?) {
    if (!Array.isArray(a)) {
        a = [a, b];
    }
    return a.reduce((p, c) => p + c);
};
```

Construct Signatures

- Represents a function that should be prefixed with “new” when invoked

```
class Person {}

var personConstructor: {
    new (name: string, age: number): Person;
};

class DerivedPerson extends Person {}

personConstructor = DerivedPerson;
```

Index Signatures

- Represents objects that can get and set elements via “[]” (arrays)

```
class Animal {}

var zoo: {
    [index: number]: Animal;
};

zoo = [ new Animal(), new Animal() ];
```

Property Signatures

- Represents gettable and settable fields
  - suffix name with “?” to indicate optional properties

```
var namedThing: {
    name: string;
    nick?: string;
};

class Person {
    constructor(public name) {}
}

namedThing = new Person('Jason');
```

Function Signatures

- Represents callable function properties
  - otherwise known as methods

```
var speakableThing: {
    speak(): void;
};

class Person {
    constructor(public name) {}
    speak() {}
}

speakableThing = new Person('Brock');
```

Array Type Literals

- Any type followed by “[]” is an array type literal
  - equivalent to index type literal returning type plus Array members

```
var people: Person[];

class Person {}

people = [ new Person(), new Person() ];
```

Function Type Literals

- Shortcut for object type literal containing a single, non-overloaded call signature

```
var add: (a: number, b: number) => number;

// Shortcut for:
// var add: { (a: number, b: number): number };

add = (a, b) => a + b;
```

Constructor Function Type Literals

- Shortcut for object type literal containing a single, non-overloaded construct signature

```
class Person {}

var personConstructor: new (name: string, age: number) => Person;

// Shortcut for:
// var personConstructor: {
//     new (name: string, age: number): Person;
// };

personConstructor = Person;
```

Interfaces

- Interfaces are named types
  - can be implemented by classes
  - compiler ensures class implements all members of interface

```
interface IPerson {
        name: string;
        speak(): void;
}

class Person implements IPerson {
        constructor(public name: string) {}
        speak() { alert('Hi!'); }
}
```

Structural Typing

- Types are considered equivalent if they have identical sets of member names and types

```
function usePerson(person: { name: string; age?: number; }) {}

class Person {
    constructor(public name: string, private age?: number) {}
}

usePerson(new Person('John'));
```

Type Inferencing

- TypeScript doesn't require annotating everything
  - can often infer types
  - inferencing usually proceeds "bottom-up"

```
// Type is: (a: any, b: any) => any
function add(a, b) {
    return a + b;
}
```

```
// Type is: (a: number, b: number) => number
function add(a: number, b: number) {
    return a + b;
}
```

Contextual Type Inferencing

- Type inferencing can sometimes proceed “top-down”

```
class Person {}

var people = [ new Person(), new Person() ];

// Type of callback function is: (p: Person) => void
people.forEach(function(p) {});
```

Dynamic Property Access

- Unknown properties can always be accessed on objects via []
  - when getting, type is “any”

```
class Person {}

var p = new Person();

p['custom'] = 42;
```

Implicit Type Conversions

- The “any” type can be implicitly converted to any other type

```
function add(a: number, b: number) {
     return a + b;
}

var data = {
    a: 1,
    b: 2
};

var a = data['a'];
var b = data['b'];

alert(add(a, b).toString());
```

Explicit Type Conversions

- When you have more knowledge than the type system, can explicitly cast from one type to the other

```
var canvas = document.getElementsByTagName('canvas')[0];
var ctx = canvas.getContext('2d');
// The property 'getContext' does not exist on value of type 'Node'
```

```
var canvas =
    <HTMLCanvasElement>document.getElementsByTagName('canvas')[0];
var ctx = canvas.getContext('2d');
```

Generics

- Functions, classes, and interfaces can be generic

```
class Pair<T1, T2> {
    constructor(private a: T1, private b: T2) {}
    first(): T1 { return this.a; }
    second(): T2 { return this.b; }
}
```

Summary

- TypeScript can help improve the developer experience while staying true to the essence of JavaScript