TypeScript

JavaScript for tools

Ben Smith ben@10consulting.com
TypeScript is a language for application-scale JavaScript development.

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.

Any browser. Any host. Any OS. Open Source.

typescriptlang.org
Design Goals

- Extend JavaScript for writing large apps (superset of JavaScript).
- Adds support for classes, interfaces & modules.
- Development tooling support
- Compiled JavaScript runs in any browser (or Node.js).
- Since JavaScript code is TypeScript code you can start off with JavaScript and just add some types here and there.
What TypeScript does is, it basically formalizes a static type system that describes JavaScript's dynamic types, but it describes them at development time.

Anders Hejlsberg
Tooling Support

- Static type checking.
- Strong type inference.
- Symbol-based navigation.
- Statement completion / intellisense.
- Code refactoring.

Currently supported in Visual Studio 2012 and Microsoft's Monaco web editor (TypeScript.org playground).

Also, text editor support for Sublime Text, EMACS, Vim.
Installation

Visual Studio 2012
Plugin available to download.

Node.js

```
npm install -g typescript
```

Provides a command-line compiler.

```
tsc source.tc
```
File Extensions

- `.ts` is the extension for source files.
- `.d.ts` is the extension for declaration files.
Declaration Source Files

- Provide type definitions, separate from the corresponding source.
- Analogous to header files in C/C++.
- Can be used to describe the exported virtual TypeScript types of a JavaScript library or module when a third-party developer consumes it from TypeScript.
- Gives type safety, intellisense and compile errors.
- DOM and jQuery provided with TypeScript.
- Write your own for any existing JavaScript library / code.
Type Annotations

- Optional static typing.
- Lightweight way to record the intended contract of a function or variable.
- Applied using a post-fix syntax.
- Ideally suited to when the `: T` is optional.

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

- Return type of the function can be inferred.
- Supports optional types via `?`
Types

- All types are subtypes of a single top type called the `Any` type.
- Represents any JavaScript value with no constraints.

Primitive Types

- `number`
- `bool`
- `string`
- `null`
- `undefined`

Object Types

- Class, module, interface and literal types.
- Supports typed arrays: `var reports: Employee[] = [];`
Arrow Function Expressions

- New feature planned for ECMAScript 6.
- Compact form of function expressions that omit the function keyword.
- Similar to lambda expressions in C#.
- Lexical scoping of `this`.
Arrow Function Expressions

- New feature planned for ECMAScript 6.
- Compact form of function expressions that omit the function keyword.
- Similar to lambda expressions in C#.
- Lexical scoping of this.

```javascript
var messenger = {
    message: "Hello World",
    start: function() {
        setTimeout(() => {
            alert(this.message);
        }, 3000);
    }
};
messenger.start();
```
Arrow Function Expressions

- New feature planned for ECMAScript 6.
- Compact form of function expressions that omit the function keyword.
- Similar to lambda expressions in C#.
- Lexical scoping of `this`.

```javascript
var messenger = {
    message: "Hello World",
    start: function() {
        setTimeout(() => { alert(this.message); }, 3000);
    }
};
messenger.start();
```

```javascript
window.onmousemove = e => {
    console.log('Mouse at (' + e.screenX + ', ' + e.screenY + ')');
}
```
Classes

- Support for ECMAScript 6 alike classes.
- Methods are translated into JavaScript prototype chain - more memory efficient than using closures with anonymous functions.
- `public` or `private` member accessibility.
- Parameter property declarations via constructor.
- Supports single-parent inheritance.
- Derived classes make use of `super` calls to parent.
class Animal {
    constructor(public name) {}
    move(meters) {
        alert(this.name + " moved " + meters + "m.");
    }
}

class Snake extends Animal {
    move() {
        alert("Slithering...");
        super.move(5);
    }
}

class Horse extends Animal {
    move() {
        alert("Galloping...");
        super.move(45);
    }
}
Classes

class Animal {
    constructor(name) {
        this.name = name;
        this.move = function(meters) {
            alert(this.name + ' travelled ' + meters + 'm');
        }
    }
}

class Snake extends Animal {
    constructor() {
        super.apply(this, arguments);
    }
    move(meters) {
        super.move(meters);
        alert('Slithering...');
    }
}

class Horse extends Animal {
    constructor() {
        super.apply(this, arguments);
    }
    move(meters) {
        super.move(meters);
        alert('Galloping...');
    }
}

var Animal = (function() {
    var __extends = this.__extends || function(d, b) {
        function __copy ctor {
            this.constructor = d;
        }
        __prototype = b.prototype;
        d.prototype = new __copy ctor();
    }
    var Animal = (function() {
        var __extends = this.__extends || function(d, b) {
            function __copy ctor {
                this.constructor = d;
            }
            __prototype = b.prototype;
            d.prototype = new __copy ctor();
        }
    }
    var Snake = (function(_super) {
        __extends(Snake, _super);
        function Snake() {
            __copy ctor.apply(this, arguments);
        }
        Snake.prototype.move = function(meters) {
            alert('Slithering...');
            super.prototype.move.call(this, meters);
        }
        return Snake;
    })(Animal);
    var Horse = (function(_super) {
        __extends(Horse, _super);
        function Horse() {
            __copy ctor.apply(this, arguments);
        }
        Horse.prototype.move = function(meters) {
            alert('Galloping...');
            super.prototype.move.call(this, meters);
        }
        return Horse;
    })(Animal);
Interfaces

- Designed for development tooling support only.
- No output when compiled to JavaScript.
- Structural type system - interfaces are automatically implemented by any object/prototype that complies structurally.
- Supports overload by parameter signature.
- Open for extension (may declare across multiple files).
- Supports implementing multiple interfaces.
Interfaces

```java
interface Drivable {
    start(): void;
    drive(distance: number): void;
    getPosition(): number;
}

class Car implements Drivable {
    private isRunning: bool = false;
    private distanceFromStart: number;

    public start(): void {
        this.isRunning = true;
    }

    public drive(distance: number): void {
        if (this.isRunning) {
            this.distanceFromStart += distance;
        }
    }

    public getPosition(): number {
        return this.distanceFromStart;
    }
}
```

*Note* interfaces have no *run-time* representation - they are purely a *compile-time* construct.
Interfaces

```javascript
var Car = (function () {
    function Car() {
        this.isRunning = false;
    }
    Car.prototype.start = function () {
        this.isRunning = true;
    };  
    Car.prototype.drive = function (distance) {
        if(this.isRunning) {
            this.distanceFromStart += distance;
        }
    };  
    Car.prototype.getPosition = function () {
        return this.distanceFromStart;
    };  
    return Car;
})();

Note interfaces have no run-time representation - they are purely a compile-time construct.
```
Structural Types

```typescript
interface Person {
    firstname: string;
    lastname: string;
}

function greeter(person: Person) {
    return "Hello, " + person.firstname + " " + person.lastname;
}

var user = {firstname: "Jane", lastname: "User"};
greeter(user);

Note that structural typing also applies to interface methods.
```
function greeter(person) {
    return "Hello, " + person.firstname + " " +
    person.lastname;
}

var user = {
    firstname: "Jane",
    lastname: "User"
};
greeter(user);

**Note** that structural typing also applies to interface methods.
Modules

- Analogous to .NET namespaces.
- Prevents global variable naming collisions.
- Closely aligned with those proposed for ECMAScript 6.
- Supports code generation targeting CommonJS and AMD module systems.
- Accessibility for internal and external modules.
- Allows exposing a public API.
interface IPoint {
    getDist(): number;
}

module Shapes {
    // Class exported for public consumption
    export class Point implements IPoint {
        // Constructor
        constructor (public x: number, public y: number) {}

        // Instance member
        getDist() { return Math.sqrt(this.x * this.x + this.y * this.y); }

        // Static member
        static origin = new Point(0, 0);
    }
}

var p: IPoint = new Shapes.Point(3, 4);
var dist = p.getDist();
```javascript
var Shapes;
(function (Shapes) {
    var Point = (function () {
        function Point(x, y) {
            this.x = x;
            this.y = y;
        }
        Point.prototype.getDist = function () {
            return Math.sqrt(this.x * this.x + this.y * this.y);
        }
    
    };Point.origin = new Point(0, 0);
    return Point;
})(Shapes || (Shapes = {}));

var p = new Shapes.Point(3, 4);
var dist = p.getDist();
```
Source File Dependencies

- Compiler automatically determines a source file's dependencies.
- Uses *reference comments* and *import* declarations.
- All references are analysed for their dependencies.
- Visual Studio uses these references to load all associated source files from a single *ts* file.
Source File Dependencies

- Compiler automatically determines a source file's dependencies.
- Uses *reference comments* and `import` declarations.
- All references are analysed for their dependencies.
- Visual Studio uses these references to load all associated source files from a single `.ts` file.

Reference comment

```bash
/// <reference path="jquery.d.ts"/>
```

Import declaration

```javascript
import log = module("log");
log.message("hello");
```
Source Map Support

- Alleviates the debugging issues that are raised by *-to-JavaScript compilers and JavaScript minifiers.
- You aren't debugging the code that you wrote.
- Source maps fixes this; it works like *magic*!
- A way to map combined / minified files back to their unbuilt state.
- TypeScript team have done the hard work for us by providing a Source Map generator in the compiler.
- Allows debugging and breakpoints from .ts files.

```
tsc -sourcemaps example.ts
```

- Requires either Chrome Canary or WebKit nightly.
- Not yet supported by Visual Studio 2012.
- ... but available via Web Essentials 2012 plugin.
Source Map Support

- Alleviates the debugging issues that are raised by *-to-JavaScript compilers and JavaScript minifiers.
- You aren’t debugging the code that you wrote.
- Source maps fixes this; it works like magic!
- A way to map combined / minified files back to their unbuilt state.
- TypeScript team have done the hard work for us by providing a Source Map generator in the compiler.
- Allows debugging and breakpoints from .ts files.

```
tsc -sourcemaps example.ts
```

- Requires either Chrome Canary or WebKit nightly.
- Not yet supported by Visual Studio 2012.
- ... but available via Web Essentials 2012 plugin.

Demo
Self hosting

- Compiler is written in TypeScript, so can compile itself to JavaScript.
- Can be hosted in any ECMAScript 3 compatible runtime.
- Can host the compiler in a web browser by referencing `typescript.js`
Hosting TypeScript Compiler


```html
<script type="text/javascript" src="typescript.js"></script>
```
2. Create an output writer

```javascript
var outfile = {
    source: "",
    Write: function (s) {
        this.source += s;
    },
    WriteLine: function (s) {
        this.source += s + "\n";
    },
    Close: function () {
    }
};
```
3. Create an instance of the TypeScript compiler

```javascript
var compiler = new TypeScript.TypeScriptCompiler(outfile);

... with compilation error handling.

compiler.parser.errorRecovery = true;
compiler.setErrorCallback(function (start, len, message, block) {
    console.log('Compilation error: ', message, '
    Code block: ', block, ' Start position: ', start, ' Length: ', len);
});
```
4. Add compilation source.

```javascript
var src = $('#source').text();
compiler.addUnit(src, '');
```

... and standard lib file.

```javascript
// libfile variable contains packed declaration file
lib.d.ts
compiler.addUnit(libfile, 'lib.d.ts');
```
5. Compile.

```javascript
compiler.typeCheck();

compiler.emit(false, function createFile(fileName) {
  console.log(outfile);
  return outfile;
});

console.log('compiled: ' + outfile.source);
```
5. Compile.

```javascript
compiler.typeCheck();

compiler.emit(false, function createFile(fileName) {
  console.log(outfile);
  return outfile;
});

console.log('compiled: ' + outfile.source);
```

Demo