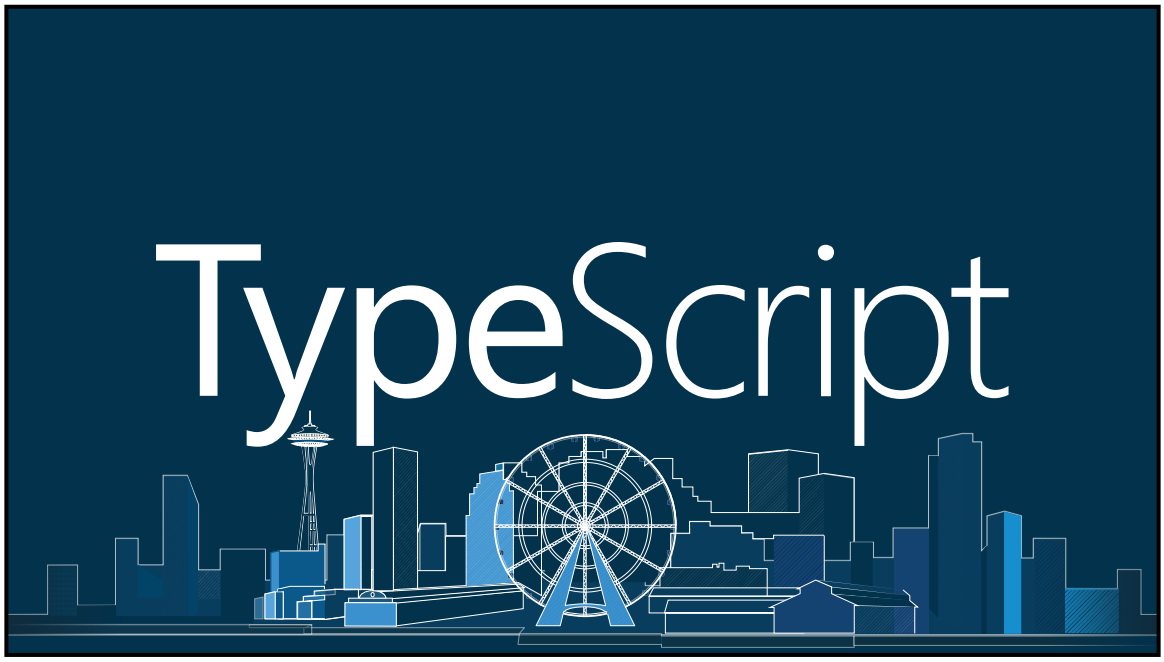


About TypeScript

- Introduced October 2012
- Open source programming language
- Maintained by Microsoft
- Apache 2 license
- Transpiles to JavaScript

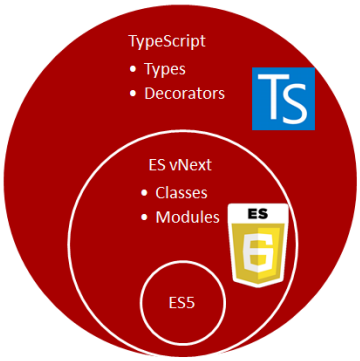
Getting started

TypeScript can be installed straight from `npm`

```
npm install --global typescript
tsc --init
tsc -w
```

- The `tsc --init` command will generate a `tsconfig.json` file
  - Represents your typescript project.
- The `tsc` command will compile your files.
- Add typescript files using the `.ts` extension

What is TypeScript



- Everything of todays JavaScript (ES5)
- Everything of the next version of JavaScript (ES6 and beyond)
- Additional features: Types, decorators

Features

To work with Angular, we'll need to know some features.

ES6 features

- Variable declarations
- Modules
- Arrow functions
- String interpolation

TypeScript features

- Types
- Classes
- Interfaces
- Structural subtyping
- Decorators

Variable declaration

- `let` declares a block-scoped variable
- `const` declares a readonly variable

```
const a = 3;
a = 4; // Error, readonly property

for (let i = 0; i < b.length; i++) {
    let y = b[i];
}
console.log(y); // Error, cannot find name 'y'

let callbacks = [];
for (let i = 0; i <= 2; i++) {
    callbacks[i] = function () { return i * 2; };
}

callbacks[0]() === 0;
callbacks[2]() === 4;
```

Always use `const` or `let` instead of `var`

Modules


```
// lib/math.js
export function sum (x, y) {
    return x + y;
};
export let pi = 3.141593;

// someApp.js
import * as math from './lib/math';
console.log('2π = ' + math.sum(math.pi, math.pi));

// otherApp.js
import { sum, pi } from './lib/math';
console.log('2π = ' + sum(pi, pi));
```

- Using `export` or `import` in a file will transform that file in an **external module**
- Declaring variables in an external module *won't* pollute the global scope

A note on modules

- Syntax is in the ECMAScript specification
- How a browser retrieves modules is **not**
  - When writing Angular we always need a module loader (SystemJS/webpack)
- An `import` can be *relative* or *non-relative*
  - Relative import: module name starts with `./` or `../`
    - Always relative to the current file
    - Example: `import * as math from './lib/math';`
  - Non-relative import: all other module names
    - Are resolved using the `node_modules` folder (because `moduleResolution` is `node`)
    - Example: `import { Component } from "angular2/core";`

Arrow functions

Use arrow functions instead of using the `function` keyword.

```
let sortMod5 = (numbers) => {
    return numbers.filter(num => num %5 === 0);
};

let User = function(name) {
    this.name = name;
    this.filter = (items) => items.filter(item => item.name === this.name);
}; // Lexical `this`

let frank = new User('Frank');
```

Using `=>` will *fix* the `this`-pointer

Rule of thumb: when in doubt, don't use `function`

String interpolation

Multiline string interpolation using back ticks `

```
let aValue: 'some string';
let n = `Code example:
  string a = '${aValue}';`;

console.log(n);
/*
Code example:
  string a = 'some string';
*/
```

Types

- Use `:` to add optional type annotations
- Build-in types: `string`, `number`, `boolean`, `Array<T>`

```
let name: string = 'Frank';

function add(x: number, y: number): number {
    return x + y;
}

let list: number[] = [2, 4, 5]; // Synonym: Array<number>

let isEven: (n: number) => boolean = (n: number) => n % 2 === 0;

name = true;   // Error: Type 'boolean' is not assignable to type 'string'
list = isEven; // Error: Type '(n: number: => boolean)'
               //   is not assignable to type 'number[]'
```

Above type annotations can all be inferred by TypeScript (except for parameter types)

Classes

A more intuitive, object oriented style classes

```
class Point {
    constructor(public x: number, public y: number) { }

    length(o: Point) {
        const width = this.x - o.x;
        const height = this.y - o.y;
        return Math.sqrt(width * width + height * height);
    }

    static origin = new Point(0, 0);
}

class ColoredPoint extends Point {
    constructor(x: number, y: number, public color: string) {
        super(x, y);
```

Relies on prototypal inheritance.

Interfaces

- Define the *shape* that a value has
- No runtime equivalent

```
interface Colored {
    color: string;
}

class ColoredPoint extends Point implements Colored  {
    color: string;
}
```

Structural subtyping

Sometimes called *duck typing*

> *If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck*

```
interface Named {
    name: string;
}
class Person {
    constructor(public name: string) { }
}

let p: Named = new Person('Frank');

const getName = (named: { name: string }) => named.name;
const pName = getName(p);
```

Decorators

Decorators can be used to add functionality at runtime

```
@Component({
    selector: 'my-app',
    template: `<h1>My application</h1>`
})
class MyAppComponent { }
```

Within Angular, they are mostly used for dependency injection using reflection.

```
console.log(Reflect.getOwnMetadata('annotations', AppComponent)[0].selector);
// 'my-app'
```