TypeScript

Sang Shin
JPassion.com
“Code with Passion!”

Topics

- JavaScript language variants
- ES6
- TypeScipt

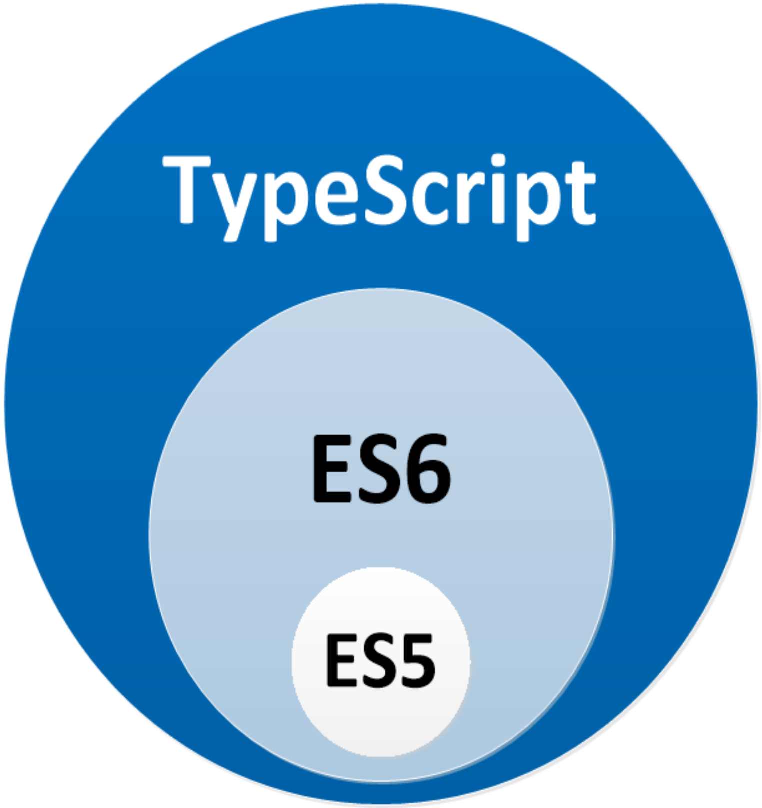
JavaScript Language Variants

JavaScript language variants

- There are many variants
  - ES5, ES6, then ES7 (ES2016) TypeScript, AtScript, Dart, CoffeeScript, ...
- ES6 and ES7 come with a set of new language features
- Today, however, ES5 is still the version that is most widely supported by browsers
- Typescript is Microsoft’s extension of JavaScript that comes with powerful type checking abilities and object oriented features
  - TypeScript is superset of ES5, ES6, ES7
  - TypeScript code gets transpiled into ES5 code for execution in browsers

TypeScript is superset of JavaScript

- Any ES5 and ES6 JavaScript programs are valid TypeScript programs
- TypeScript provides extra features such as Interfaces, Generics over ES6 (We will cover these in detail later on)



Angular 2 and TypeScript

- Angular 2 uses TypeScript as a language of choice
  - You can build Angular 2 apps using JavaScript but you lose type checking (and other language features) of TypeScript
- Most documentation and example codes in Angular 2 are based on TypeScript

TypeScript Tools

- Transpiler
  - *tsc myTypeScriptCode.ts* (to compile TS code to JS)
  - *node myTypeScriptCode.js* (to execute JS code)
- Transpiler and executor in a single step
  - *[sudo] npm install -g ts-node*
  - *ts-node myTypeScriptCode.ts* (to compile and execute TS code)

Lab: Install Typescript

- Install node.js (if it has not been installed already)
- Install typescript
  - [sudo] npm install -g typescript
  - [sudo] npm install -g ts-node
- Run typescript code
  - ts-node myCode

ES6

ES6 Features

- Classes
- *let* and *const* variables
- Arrow functions (fat arrow)
- Modules
- Promises
- Decorators
- for-of
- ...

ES5 (old-style) code of creating a Class

```
function User(id, firstName, lastName) {
  this.id = id;
  this.firstName = firstName;
  this.lastName = lastName;
}

User.prototype = {
  getFullName: function() {
    return return this.firstName + " " + this.lastName;
  }
};
```

ES6: Class

```
class User {
    id;
    firstName;
    lastName;

    constructor (id, firstName, lastName){
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(){
        return this.firstName + " " + this.lastName;
    }
}

let user1 = new User(1, "sang", "shin");
console.log(user1.getFullName());
```

ES6: Class Inheritance

```
class Person {
   ...
}

class Student extends Person {
    school;

    constructor (firstName, lastName, school){
       super(firstName, lastName);
       this.school = school;
    }

    getFullNameInfo(){
        return this.firstName + " " + this.lastName + " "
               + this.school;
    }
}
```

ES6: Variables – var, let, const

- ES6 provides a new way of specifying variables: let and const
  - *let* and *const* create block scoped variables – they live and die within {..} block
  - *let* can be reassigned while *const* variable can't be reassigned
- Before ES6, we only had var which create a function-scoped variable

```
// ES5 example - “i” will be available after the loop
for (var i in items) {
}

// ES6 example - “i” will not be available after the loop
for (let i in items) {
 // i is available
}
```

ES6: Template Strings

```
// - write long inline string without having to use concatenation
// - defined opening and closing back ticks
let template1 = `
  <div>
    <h2>Rufferford's Travels</h2>
    <p>
      A most gripping tale of one dog's quest
      for more flavors.
    </p>
  </div>
`;

// You can also do string interpolation using
// ${expression} placeholders:
let x = 5;
let y = 10;
let template2 = `
  <div>The sum is ${ x + y }</div>
`;
```

ES6: JavaScript Modules

- ES6 standardized module system (over two existing module systems – AMD and CommonJS)
- By default, anything you declare in a file in a ES6 project is not available outside that file. You have to use the export keyword to explicitly make it available
- Not the same thing as Angular module system

```
// teacher.ts
export class Teacher {
  ...
}

// main.ts
import { Teacher } from './teacher';

let teacher: Teacher = new Teacher('sang');
console.log(teacher.getName());
```

ES6: Promises

- Promises make it easier to write asynchronous code compared to using callbacks

```
let myPromise = new Promise(
    (resolve, reject) => {
        setTimeout(() => resolve("JPassion.com"), 3000);
    });

myPromise.then(value => console.log(value))
                  .catch(error => console.log(error));
```

ES6: Arrow Functions (Fat Arrow)

- More concise syntax for writing function expressions – no need to type the function keyword, return keyword (it’s implicit in arrow functions), and curly brackets

```
// ES5
var multiply1 = function (x, y) {
  return x * y;
}


// ES6 using Fat arrow
var multiply2 = (x, y) => x * y;
```

ES6: Arrow Functions (Fat Arrow)

- Fat arrow also changes the way “this” binds in functions
- Problem (when Fat Arrow is not used)
  - In JavaScript, each function in JavaScript defines its own “this” context object
  - If the function is a callback function, “this” does not represents the context you want – one workaround is to create a closure as shown below

```
class MyClass5 {

  name: string = "Sang5";

  constructor() {
    var self = this; // create a closure
    setTimeout(function () {
      console.log(self.name); // use a closure as work-around
    }, 3000);
  }
}
```

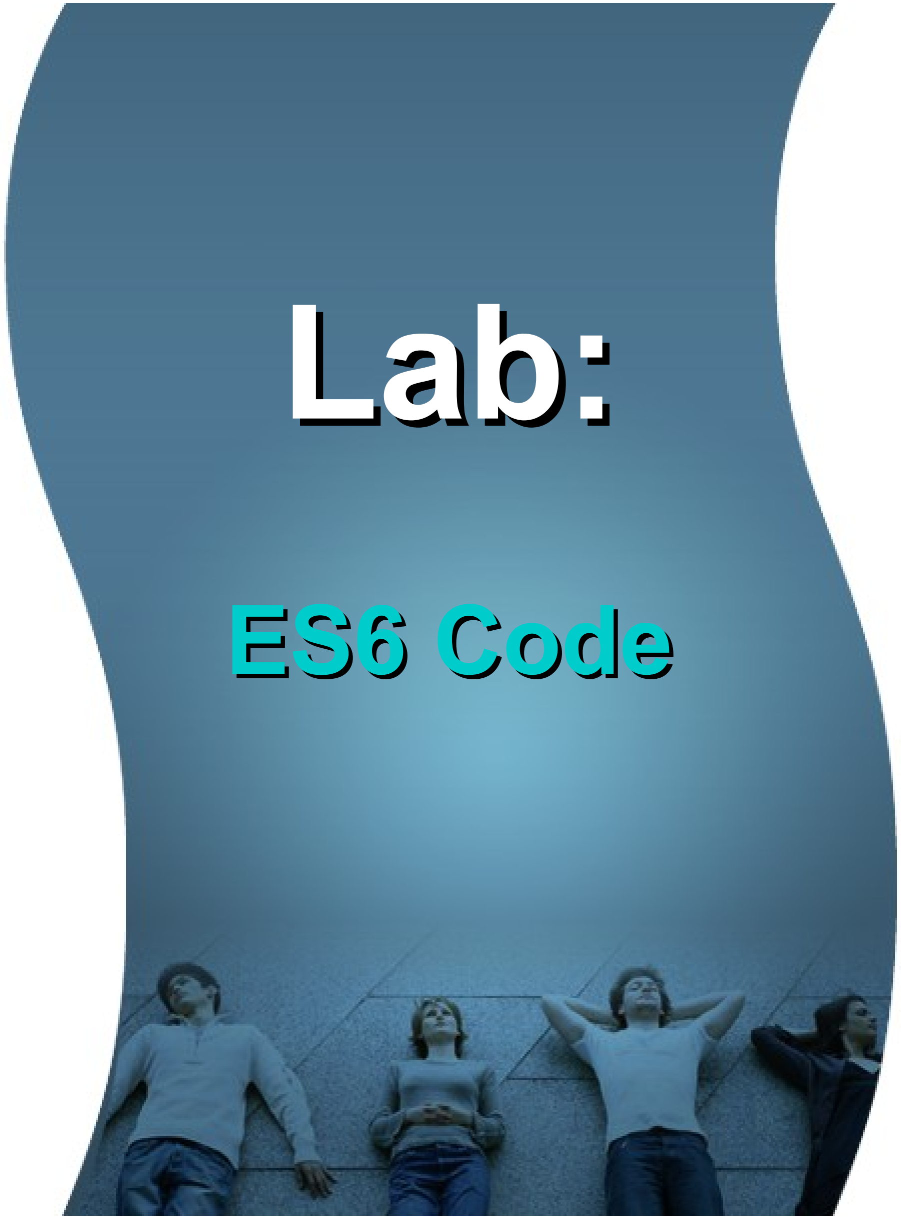
ES6: Arrow Functions (Fat Arrow)

- Fat arrow does not create its own “this” context object - so there is no need to use a workaround such as using a closure, instead you can use this

```
class MyClass {

  name:string;

  constructor() {
    console.log("Expect Sang in 3 seconds");
    this.name = 'Sang';
    setTimeout(() => {
      console.log(this.name);
    }, 3000);
  }
}
let myClass = new MyClass();
```



Lab:

ES6 Code

TypeScript

Why TypeScript?

- Building large-scale JavaScript application without using compile-type checking turned out to be very challenging
  - Even with all the testings you can do

- Building large-scale code without proper tooling such as compile time error detection, refactoring capabilities, code completion, etc turned out to be very challenging as well
  - JavaScript tools are not powerful enough compared to the ones in other OO programming languages (Java, C#)

- TypeScript is to the rescue
  - TypeScript is a strongly-typed language, which enables compile-time type checking and availability of tools

TypeScript Provided Features over ES6

- Type annotations with Compile-time type checking
- Public/Protected/Private (for controlled access)
- Type inference
- Interfaces
- Generics
- Decorators
- ...

TypeScript code with Compile-time Types

```
class User {
  id: number;
  firstName: string;
  lastName: string;

  constructor(id: number, firstName: string, lastName: string) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
  }
  getId() {
    return this.id;
  }
  getFirstName(): string {
    return this.firstName;
  }
  setFirstName(firstName: string) {
    this.firstName = firstName;
  }
  setLastName(lastName: string) : void {
    this.lastName = lastName;
  }
}
```

Public/Protected/Private (for controlled access)

```
class Student2 {
    private name: string;
    protected hobby: string;
    public age: number;   // default

    public setName(name: string){
        this.name = name;
    }

    getName(): string {
       return this.name;
    }
}
```

Interfaces

```
interface User {
    username: string;
    password: string;
    confirmPassword?: string; // Optional property
}

let user:User;

// This value does not satisfy the interface => Compilation error
// user = { anything: 'anything', anynumber: 5};

// These values do satisfy the interface
user = {username: 'sang', password: 'xyz', confirmPassword: 'xyz'};
user = {username: 'sang', password: 'xyz'};
```

Interfaces

```
// Interfaces can also contain functions (without the function body
// as it is a blueprint/ requirement)

interface CanDrive {
    sayGreeting: (message: string) => number;
    accelerate(speed: number): void;
    brake(): string;
}

let car: CanDrive = {
    // sayGreeting: function (message) {
    //     return message.length;
    // },
    sayGreeting: (message) => message.length,
    accelerate: function (speed: number) {
    },
    brake: function () {
        return "Code with Passion!";
    }
};
```

Generics

```
let numberArray: Array<number>; // will only accept numbers

// Try to initialize it with strings

numberArray = ['test']; // => Error
numberArray = [1,2,3];
```

Decorators

- Decorators are functions that are invoked with a prefixed @ symbol, and immediately followed by a class, parameter, method or property
- Decorators are proposed for a future version of JavaScript, but the Angular 2 team really wanted to use them, and they have been included in TypeScript

Lab:

TypeScrpt Code

Code with Passion!
JPassion.com