Modern Web Development for Java Programmers

Unit 2. Becoming Productive with JavaScript. Intro to TypeScript. AngularJS Basics.

Unit 2 Timeline

- Overview of JavaScript alternatives 5 min
- TypeScript Basics 30 min
- *Walkthrough 1:* TypeScript in Action 10 min
- Overview of JavaScript frameworks 10 min
- AngularJS Basics 30 min
- *Walkthrough 2, 3:* First AngularJS app 10 min
- More on AngularJS 30 min
- *Walkthrough 4:* AngularJS with TypeScript and Grunt 25 min
- Configuring IntelliJ IDEA environment 10 min
- *Walkthrough 5:* Developing Home page and Product Details page 20 min

Overview of JavaScript Alternatives

JavaScript Pros and Cons

Pros:

- Executes natively:
  - Easy to debug
  - Performance
- Huge set of libraries
- Huge community
- Well-know by developers

Cons:

- Wired behaviour
- No structure
- No types
- Inconsistent patterns
- Bad tooling (relatively)

JavaScript Alternatives

- Scala.js
- ClojureScript
- Dart
- CoffeeScript
- **TypeScript**

TypeScript vs JavaScript: Pros

Pros:

- ~~Executes natively~~ **Compiles to JavaScript**:
  - Easy to debug **(with source maps)**
  - Performance **(identical)**
- Huge set of libraries **(100% compatible with JavaScript)**
- Huge community
- Well-know by developers

TypeScript vs JavaScript: Cons

Cons:

- ~~Wired behaviour~~ **Hides away (e.g.‘use strict’ and === by default)**
- ~~No structure~~ **(modules, classes, exports)**
- ~~No types~~ **(optional type annotations)**
- ~~Inconsistent patterns~~ **(modules, classes)**
- ~~Bad tooling~~ **(decent static analyzer)**

TypeScript Features

- Superset of JavaScript, 100% compatible
- Compiles to semantic JavaScript
- Supports modules and classes
- Supports type annotations
- Aims to be as close to ES6 as possible

TypeScript: Types

JS:

No explicit typing

TS:

- any - explicitly untyped
- string
- number
- boolean
- void, null, undefined - special types

TypeScript: Declaring variables

JS:

```
var name = 'John Smith';
```

TS:

```
var name = 'John Smith';
var name: string = 'John Smith';
```

TypeScript: Declaring functions

JS:

```
// Statement:
function getName() {
    return 'John Smith';
}

// Expression:
var getName = function () {
    return 'John Smith';
};
```

TS:

```
// Single expression:
var getName = () => 'John Smith';

// Multiple expressions:
var getName = () => {
    var name = 'John Smith'.toUpperCase();
    return name;
};

// Type annotations and default values:
var getName = (toUpper: boolean = true) => {
    var name = 'John Smith';
    return toUpper
        ? name.toUpperCase()
        : name;
};
```

TypeScript: Declaring classes

JS:

```
// Constructor function and fields:
function Person(name, title) {
    this.name         = name;
    this.subordinates = [];
}
```

TS:

```
// Declarative class definition:
class Person {
    name        : string;
    subordinates: Array<Person> = [];
}

// Short version:
class Person {
    constructor(
        public name: string,
        public subordinates: Person[] = [])
}
```

TypeScript: Instance methods

JS:

```
Person.prototype.addSubordinate =
  function (person) {
      this.subordinate.push(person);
  };
```

TS:

```
class Person {
    addSubordinate(p: Person): void {
        // `this` is LEXICALLY scoped
        this.subordinates.push(p);
    }
}
```

TypeScript: Static members

JS:

```
// Accessible only directly on `Person`
Person.GENDER = {
    'MALE'  : 0,
    'FEMALE': 1
};
```

TS:

```
class Person {
    // Accessible on any subclass:
    static GENDER = {
        'MALE'  : 0,
        'FEMALE': 1
    };
}
```

TypeScript: Getters/setters

JS:

```
// Only since ES5
Object.defineProperty(
  Person.prototype, "title", {
    get: function () {
        return this._title;
    },
    set: function (val) {
        this._title = val;
    },
    enumerable: true,
    configurable: true
});
```

TS:

```
class Person {
    // Private only for TypeScript:
    private _title: string;
    get title() { return this._title }
    set title(val: string) {
        this._title = val;
    }
}
```

TypeScript: Inheritance

JS:

```
function Person(name, title) {
    this.name  = name;
    this.title = title;
}
function Employee(company, department) {
    this.company    = company;
    this.department = department;
}
Employee.prototype = new Person();
```

TS:

```
class Person {
    constructor(public name : string,
                public title: string) {}
}
class Employee extends Person {
    constructor(public company   : string,
                public department: string) {}
}
```


TypeScript: Interfaces

JS:

Doesn’t support in any form

TS:

```ts
interface IAddress {
    country: string;
    zipcode: string;
}

interface IPerson {
    name: string;
}

class Person implements IPerson {
    name: string; // Required by interface
    addresses: Array<IAddress> = [];
}
```


TypeScript: Anonymous types

JS:

Doesn’t support in any form

TS:

```
class BidResponse {
    payload  : { itemId: number; bid: number };
    callbacks: Array<(resp: BidResponse) => void>;
}
```

TypeScript: Modules

JS:

```
var myNamespace = (function () {
    var myPrivateVar, myPrivateMethod;

    // A private variable
    myPrivateVar = 0;

    // A private function
    myPrivateMethod = function (foo) {
        console.log(foo);
    };

    return {
        // A public variable
        myPublicVar: "foo",

        // A public function utilizing privates
        myPublicFunction: function (bar) {
            // Increment our private counter
            myPrivateVar++;
            // Call our private method using bar
            myPrivateMethod(bar);
        }
    };
})();
```

TS:

Internal modules:

```
module myapp {
    export module models {
        export class Person {}
    }
}
// Fully-qualified form:
var p1 = new myapp.models.Person();
// Shorthand:
import m = myapp.models;
var p2 = new m.Person();
```

External modules:

```
// models.ts
export Person {}

// app.ts
var models = require('./models');
```

TypeScript Definitions

- A.k.a ambient declarations.

- Introduces a variable into a TypeScript, but results in no JavaScript code and has zero impact for generated program.

```
interface JQuery {
    text(content: string);
}

interface JQueryStatic {
    get(url: string, callback: (data: string) => any);
    (query: string): JQuery;
}
declare var $: JQueryStatic;
```

- Useful for stubbing 3rd-party library's code and make TypeScript compiler happy.

- We can create custom definition files.

TypeScript Definitions

- There is a huge collection of TypeScript definitions libraries - DefinitelyTyped (https://github.com/borisyankov/DefinitelyTyped)

- DefinitelyTyped available in *bower.*

- DefinitelyTyped available in IntelliJ Idea: *Preferences* → *Project Settings* → *JavaScript Libraries* → *Download…* → *TypeScript Community stubs*.

- We will use both repositories: bower - to make tsc happy, JavaScript Libraries - to make Idea happy.

Walkthrough 1

- Start IntelliJ IDEA and create a new empty project according to the document import_code_manual.pdf.

- Ensure the File Watcher plugin is properly set up (according to the instructions sent before the class).

- Right-click on w1.html from the directory walkthroughs/w1/ and select Open in Browser (Google Chrome has to be your default Web browser).

- In Chrome browser open Developer Tools from the Chrome's menu View | Developer.

- Open tab Console, ensure the program works properly and prints two lines with computed taxes without any errors.

- In IntelliJ Idea open file w1.ts and refactor the program in a way to be written in the TypeScript style (type annotations, using classes, instance members, etc.).

- Ensure the program still works correctly in Chrome browser.

- Compare your implementation with original JavaScript code and implementation provided in w1-solution.ts file. Think about writing your code in a more structured way.

Walkthrough 1: JS version

```
// Convert this code to a TypeScript version.
function Person(name) {
    this.name = name;
}
Person.prototype.doTaxes= function(){
    var taxDeduction = 500;

    //private function
    function mafiaSpecial(income){
        return income*0.05 - taxDeduction*2;
    }
    //exposed function
    return function(income) {
        var yourTax;

        if (this.name != "God Father") {
            yourTax = income * 0.05 - taxDeduction;
        } else {
            yourTax = mafiaSpecial(income);
        }
        console.log("My dear " + this.name + ", your tax is " + yourTax);
        return yourTax;
    }
}();

var p1=new Person("John Smith");
p1.doTaxes(100000);

var p2= new Person("God Father");
p2.doTaxes(100000);
```

Walkthrough 1: TS version

```
class Person {
    private TAX_DEDUCTION: number = 500;
    private TAX_RATE: number = 0.05;

    constructor(public name: string) {
    }

    private mafiaSpecial = (income: number) =>
        income * this.TAX_RATE - this.TAX_DEDUCTION * 2;

    doTaxes(income: number): number {
        var yourTax = this.name != 'God Father'
            ? income * this.TAX_RATE - this.TAX_DEDUCTION
            : this.mafiaSpecial(income);

        console.log('My dear ' + this.name + ', your tax is ' + yourTax);
        return yourTax;
    }
}

var p1 = new Person('John Smith');
p1.doTaxes(100000);

var p2= new Person('God Father');
p2.doTaxes(100000);
```

Overview of JavaScript Frameworks

Why use a framework?

- Effectively deal with cross-browser compatibility
- Focus on the business task
- Common well-know application structure

What to choose?

Two types of frameworks:

- Feature complete
- Lightweight

Feature Complete Frameworks

- Work best for back-office applications
- Include all you might need
- Usually have better tooling
- Faster development cycle
- Hard to customize

Examples:

- Ext JS/Sencha Touch
- Dojo
- YUI

Lightweight Frameworks

- Work best for consumer-oriented public websites
- Offer narrowed set of features
- Often depend on 3rd-party libraries
- Often not really lightweight
- More flexible

Examples:

- **AngularJS**
- Backbone.js
- Ember
- Knockout

AngularJS

HTML enhanced for web apps!

What AngularJS Offers?

- “HTML enhanced for web apps!” - directives and data bindings unobtrusively enriches HTML markup
- Model View Whatever - follow the pattern that works best for your app
- Lightweight, yet pluggable with well-defined module architecture

Minimal AngularJS App

- Single HTML file:

```
<!DOCTYPE html>

<!-- STEP 2: Bootstrap AngularJS -->
<html ng-app>
<head lang="en">
    <meta charset="UTF-8">
    <title>Unit 2. Walkthrough 2.</title>
</head>
<body>
    <!-- STEP 3: Define a new variable in the root scope. -->
    <p>Type here: <input type="text" ng-model="message"/></p>

    <!-- STEP 4: Add data binding to the "message" variable. -->
    <p>Should appear: <strong>{{ message }}</strong></p>

    <!-- STEP 1: Add AngularJS dependency. -->
    <script src="angular.js"></script>
</body>
</html>
```

Defines the scope of Angular app
Not necessarily on <html>

Creates a variable on the current scope

Binds to an existing variable on the scope

Walkthrough 2

- Open the project created in the walkthrough 1.
- Right-click on w2.html from the directory walkthroughs/w2/ and select Open in Browser (Google Chrome has to be your default Web browser).
- Type something into the text input field. Check that typed message magically appears below the input field.

Angular JS App Major Players

- **Modules** - each app is a module (at least one)
- **Controllers** - handle user interactions, orchestrate models and services
- **Directives** - attach custom behavior to DOM elements, used for decomposing UI.
- **Filters** - formats expression's value

AngularJS App, Walkthrough 3 (next slide)

```
<!DOCTYPE html>
<html ng-app="auction">
<head lang="en">
    <meta charset="UTF-8">
    <title>Unit 2. Walkthrough 3.</title>
</head>
<body ng-controller="IndexController">
    <p>Type here: <input type="text" ng-model="model.message"/></p>
    <p>Should appear: <strong>{{ model.message | uppercase }}</strong></p>

    <script src="angular.js"></script>
    <script src="w3.js"></script>
</body>
</html>
```

Directive

Filter

Expression

```
var auction = angular.module('auction', []);

auction.controller('IndexController', function ($scope) {
    $scope.model = {
        message: 'Initial message'
    };
});
```

Module

Registering controller in DI container

Walkthrough 3

- Install TypeScript compiler with executing following command in your console: *npm install -g typescript*. Ensure *tsc* command is available on your path.

- Configure IntelliJ Idea's File Watchers plugin to compile *.ts into *.js. Leave all the settings with default values.

- Open the project created in the walkthrough 1.

- Right-click on w3.html from the directory walkthroughs/w3/ and select Open in Browser (Google Chrome has to be your default Web browser).

- Type something into the text input field ensure everything still works.

- Review and make sure TypeScript version of AngularJS app works the same way as the one created in walkthrough 2.

AngularJS Modules Features

- Help to structure the code
- Enable creating of reusable modules (libraries)
- Provide declarative way to bootstrap the app (no “main” method) → modules can be loaded in any order (or parallel)
- Unit-tests can load only modules they actually test or e.g. load mocks as separate module to override default behaviour.

How To Use AngularJS Modules

```
// Creates a module
var auction = angular.module('auction', []);
```

```
// Tries to return existing module.
var auction = angular.module('auction');

// Tries to return existing module.
var auction = angular.module('auction', ['ngRoute']);
```

```
<script src="angular-route.js"></script>
```

AngularJS Dependency Injection

- In-depth on Dependency Injection here

- In short: helps to avoid directly referencing concrete implementations and explicitly creating object instances

- Injects objects, doesn't help with deferred modules loading.

- Registeres and injects objects **by name**

AngularJS DI: Injecting Dependencies

1. Short form, doesn't work after minifying:

```
angular.module('auction')
    .controller('SearchController', function ($scope) {
            //...
    });
```

Renamed while minifying

$ prefix reserved for AngularJS services

2. Long form, works well in production (convenient for TypeScript):

```
var SearchController = function ($scope) {
    //..
};
SearchController['$inject'] = ['$scope'];
angular.module('auction').controller('SearchController', SearchController);
```

Constructor function's attribute, not of the instance

3. The same as 2, but using inline annotation form

```
angular.module('auction')
    .controller('SearchController', ['$scope', function ($scope) {
            //...
    }]);
```

The order matters

AngularJS DI: Registering Dependencies

- Most of them differ by type of instantiating registered object
- Every registered object belongs to a single module
- All registered services are **singletons**
- Two phases: **configuration** (constant, provider, config) and **run** (all other)

```
angular.module('auction')
    .value(/* ... */)
    .constant(/* ... */)
    .service(/* ... */)
    .factory(/* ... */)
    .provider(/* ... */)
    .controller(/* ... */)
    .directive(/* ... */)
    .filter(/* ... */)
    .animation(/* ... */)
    .config(/* ... */)
    .run(/* ... */);
```

Services

Special objects

Hooks

DI: Services

AngularJS DI: value()

- Registers a static value.
- Available in the *run* phase

This name is used to register in Angular's DI container.
It can be used for injection in other places.

```
// JavaScript version
angular.module('auction')
    .value('securityToken', '0123456789')
    .controller('LoginController', ['securityToken', function (token) {
        //...
    }]);

// TypeScript version
class LoginController {
    static $inject = ['securityToken'];
    constructor(token: string) {
        //...
    }
}
angular.module('auction').controller('LoginController', LoginController);
```

Order matters, names can be different

Leverage static members

AngularJS DI: constant()

- Similar to value(), but available in the *configuration* phase

```
// JavaScript version
angular.module('auction')
    .constant('locales', ['en-US', 'fr-CA'])
    .config(['LocalizationServiceProvider', 'locales',
      function (provider, locales) {
         provider.setSupportedLocales(locales);
      }]);

// TypeScript version
angular.module('auction')
    .constant('locales', ['en-US', 'fr-CA'])
    .config(['LocalizationServiceProvider', 'locales',
      (provider, locales) => {
         provider.setSupportedLocales(locales);
      }]);
```


AngularJS DI: service()

- Instantiated with **new**
- Registered object must be a contractor function
- Example:

```
// JavaScript version
angular.module('auction')
    .service('AuthenticationService', function () {
        //...
    });

// TypeScript version
class AuthenticationService {
    static $inject = [
        '$http'
    ];
}
angular.module('auction')
       .service('AuthenticationService', AuthenticationService);
```


AngularJS DI: factory()

- Factory must be a function that will be **invoked** to get an instance of the service:

- Use to hide private computations:

```
// JavaScript version
angular.module('auction')
    .factory('CacheFactory', function () {
        var cache = {};
        return {
            add: function (key, value) {},
            getByKey: function (key) {}
        };
    });

// TypeScript version
class CacheFactory {
    private static cache: { [key: string]: any } = {};
    add(key: string, value: any) {}
    getByKey() {}
}
angular.module('auction').factory('CacheFactory', () => new CacheFactory());
```

Completely protected, can be accessed by the factory only

TypeScript compiler will check *cache* is properly used

AngularJS DI: factory()

- Use to return constructor function and repeatedly create new instances:

```
// JavaScript version
angular.module('auction')
    .factory('ProductModel', function () {
        return function (id, price) {
            this.id = id;
            this.price = price;
        }
    });
// Usage:
angular.module('auction')
    .controller('SearchController',
        function (ProductModel) {
            var product = new ProductModel();
        });
```

Use new to create instances

```
// TypeScript version
class ProductModel {
    constructor(
        public id: number,
        public price: number) {}
}
angular.module('auction')
    .factory('ProductModel', () => ProductModel);
// Usage:
class SearchController {
    static $inject = ['ProductModel'];
    constructor(ProductModelFactory) {
        var product =
            <ProductModel>(new ProductModelFactory());
    }
}
```

Not really convenient in TypeScript

AngularJS DI: provider()

- Similar to factory, but allows configuring provider on application startup:

```
angular.module('auction')
    .provider('AuthenticationService', function () {
        this.authType;
        this.$get = function () {
            if (authType === 'basic') return new BasicAuthenticationService();
            if (authType === 'forms') return new FormsAuthenticationService();
            return new BasicAuthenticationService();
        };
    });

angular.module('auction', [])
    .config(['AuthenticationServiceProvider',
        function (authProvider) {
            authProvider.authType = 'forms';
        }]);

angular.module('auction')
    .controller('LoginController', ['AuthenticationService', function (authService) {
        authService.login();
    }]);
```

Configurable property

Notice name changes

Available on application startup

- E.g. $routeProvider allows configuring supported URLs

AngularJS DI: provider()

- Other factory methods are just syntactic sugar implemented on top of provider:

```
provider.service = function(name, Class) {
    provider.provide(name, function() {
        this.$get = function($injector) {
            return $injector.instantiate(Class);
        };
    });
}

provider.factory = function(name, factory) {
    provider.provide(name, function() {
        this.$get = function($injector) {
            return $injector.invoke(factory);
        };
    });
}

provider.value = function(name, value) {
    provider.factory(name, function() {
        return value;
    });
};
```

DI: Special Objects

AngularJS DI: controller()

- Registered objects available for **ngController** and **routing**.
- Controller **must** be a constructor function (i.e. instantiated using **new**)
- Unlike services, controllers are **not singletons**

```
// JavaScript version
angular.module('auction')
    .controller('SearchController', ['$scope', function ($scope) {
          //...
    }]);

// TypeScript version
class SearchController {
    static $inject = [
        '$scope'
    ];
}
angular.module('auction').controller('SearchController', SearchController);
```

TypeScript leverages static members

AngularJS DI: directive()

- Uses factory() underneath
- Registers special AngularJS object - directive
- Can have dependencies

Function name is not required, but convenient for debugging - readable names instead of anonymous functions in stack trace

```
angular.module('auction').directive('languageSwitcher',
    ['locales', function languageSwitcherDirectiveFactory(locales) {
        // directive definition object, mandatory AngularJS API
        return {
            restrict: 'E',
            link: function($scope, $element) {
                $element.text('Choose language: ' + locales.join(', '));
            }
        }
    }]);

<language-switcher></language-switcher>
```

AngularJS DI: filter()

- Uses factory() underneath
- Registers special AngularJS object - filter
- Can have dependencies

Notice: returns function that invoked each time the filter is applied

```
angular.module('auction').filter('join', function joinFilterFactory() {
    return function joinFilter(array, separator) {
        return array.join(separator);
    };
});

<p>{{ model.supportedLocales | join:', ' }}</p>
```

DI: Hooks

AngularJS DI: config()

- Use to configure providers at configuration phase
- Can have only provider and constant dependencies

```
angular.module('auction', [])
    .config(['AuthenticationServiceProvider',
        function (authProvider) {
            authProvider.authType = 'forms';
        }]);
```

AngularJS DI: run()

- Runs after $injector is created, at the beginning of run phase
- Use for application initialization logic, e.g. global events binding, auto-login, geo-location.

```
angular.module('auction').run(['GeoService', function (geoService) {
    geoService.determineLocation();
}]);
```


AngularJS Controllers

Revisiting Controllers

- Handle user interactions
- Orchestrate models and services
- *Provide data-binding source (scope) for views*
- *Receive control when a route is triggered*

How to use controller

```
class MainController {
  static $inject = ['$scope'];
  constructor($scope) {
    $scope.awesomeThings = [
      'HTML5 Boilerplate',
      'AngularJS',
      'Karma'
    ];
  }
}
angular.module(‘auction').controller('MainCtrl', MainController);
```

Scope shares variables between view and controller

```
<div ng-controller=“MainCtrl”>
  <ul>
    <li ng-repeat="thing in awesomeThings">{{ thing }}</li>
  </ul>
</div>
```

AngularJS Scopes

- Scope shares variables between view and controller
- Scope is the **only** source for data-binding
- Root scope implicitly created for the entire app
- Child scopes are created for controllers and directives (depends on configuration).
- If a data-binding target isn't found on the child scope, AngularJS checks ancestors chain up to the root scope. Similar to prototypal inheritance chain.

Most frequently used AngularJS services

- $scope - access to the current scope
- $rootScope - access to the root scope
- $http - HTTP requests
- $location - to work with window.location
- $q - access to promise/deferred API
- $templateCache - to cache views/directive templates
- $window - to access window object

Additional Resources

- AngularJS Developer Guide - http://docs.angularjs.org/guide
- Collection of articles - http://www.ng-newsletter.com/posts/
- Collection of AngularJS learning resources - https://github.com/jmcunningham/AngularJS-Learning

Homework 2

- Using Yeoman start a new AngularJS version of auction app. Reproduce steps from unit's 2 v0, v1 and v2 versions of the app: clean up generated project, enable TypeScript support.

- Using AngularJS, JavaScript, and Bootstrap framework reproduce the Search Results Web page developer in the homework 1 based on the provided mockup in the file named *SearchResultsMockup.png*. Attach the click event handler to the button Search so it'll open the Search Results page. Hint: use hg-click and controller's instance methods to bind to the click event.

- Send your homework (2 URLs: link to GitHub repository with the source code and link to the app deployed to GitHub Pages) before the next lesson to training@faratasystems.com. **Important. The subject of your email should start with the word Homework.**

- Post questions at your Google group by sending emails at modernwebfeb2014@googlegroups.com and help each other.

- Use chapters additional resources provided in the Unit 2 slides.