

Introduction to TypeScript. Building and Deploying TypeScript Project

Trayan Iliev

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: <http://www.iproduct.org>

Oracle®, Java™ and JavaScript™ are trademarks or registered trademarks of Oracle and/or
Microsoft .NET, Visual Studio and Visual Studio Code are trademarks of Microsoft Corp

Agenda I

1. Introduction to some ECMAScript – ES 6/7/8 new features
2. TypeScript Hello World
3. Configuring, building and deploying TypeScript project – node package manager (**npm**), package.json, tsconfig.json, **@types** and npm packages, TypeScript compiler options.
4. Linting TypeScript code – **tslint.json**
5. Module resolution. Configuring **System.js** module loader.
6. Introduction to TypeScript – functions, interfaces, classes and constructors.
7. Common types: Boolean, Number, String, Array, Enum, Any, Void. Duck typing.

Agenda II

- 8. Declaring contracts using Interfaces – properties and optional properties, function types, class types, array types, hybrid types, extension of interfaces.
- 9. Classes – constructors, public/private properties, get and set accessors, static and instance sides.
- 10. Functions and function types – optional default and rest parameters, function lambdas and use of this, overloads.
- 11. Using Enums.
- 12. Modules in TypeScript – namespaces and modules (former internal and external modules), using import and export

Agenda III

- 13. Interoperability with external JS libraries – ambient type declarations and ambient modules, typings, @types.
- 14. Generic type parameters – writing generic functions and classes, generic constructors, bounded generics.
- 15. Advanced types. Symbols. Declaration merging. Decorators. Using mixins in TypeScript.
- 16. Production grade integration with build tools – **npm**, **webpack**.

Where is The Code?

Angular and TypeScript Web App Development
code is available @GitHub:

<https://github.com/iproduct/course-angular>

Brief History of JavaScript™

- JavaScript™ created by Brendan Eich from Netscape for less than 10 days!
- Initially was called Mocha, later LiveScript – Netscape Navigator 2.0 - 1995
- December 1995 Netscape® и Sun® agree to call the new language JavaScript™
- “JS had to 'look like Java' only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JS would have happened.”



B. E. (<http://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1021>)

The Language of Web

- JavaScript™ success comes fast. Microsoft® create own implementation called **JScript** to overcome trademark problems. JScript was included in Internet Explorer 3.0, in August 1996.
- In November 1996 Netscape announced their proposal to **Ecma International** to standardize JavaScript → **ECMAScript**
- JavaScript – most popular client-side (in the browser) web programming language („de facto“ standard) and one of most popular programming languages in general.
- Highly efficient server-side platform called **Node.js** based on **Google V8 JS engine**, compiles JS to executable code Just In Time (JIT) during execution (used at the client-side also).

Object-Oriented JavaScript

Three standard ways to create objects in JavaScript:

- Using **object literal**:
`var newObject = {};`
- Using **Object.create(prototype[, propertiesObject])** (prototypal)
`var newObject = Object.create(Object.prototype);`
- Using **constructor function** (pseudo-classical)
`var newObject = new Object();`

Object Properties

- Object-Oriented (OO) – object literals and constructor functions
- Objects can have named properties

Ex.: `MyObject.name = 'Scene 1';`
`MyObject['num-elements'] = 5;`
`MyObject.prototype.toString = function() {`
`return "Name: " + this.name + ": " + this['num-elements'] }`

- Configurable object properties – e.g. read only, get/set, etc.

Ex.: `Object.defineProperty(newObject, "someKey", {`
`value: "fine grained control on property's behavior",`
`writable: true, enumerable: true, configurable: true`
`});`

Property Getters and Setters

Ex.: `function PositionLogger() {
 var position = null, positionsLog = [];
 Object.defineProperty(this, 'position', {
 get: function() {
 console.log('get position called');
 return position;
 },
 set: function(val) {
 position = val;
 positionsLog.push({ val: position });
 }
 });
 this.getLog = function() { return positionsLog; };
}`

JavaScript Features

- The state of objects could be changed using JS functions stored in object's **prototype**, called **methods**.
- Actually in JavaScript **there were no real classes**, - only objects and constructor functions before ES6 (ES 2015, Harmony).
- JS is **dynamically typed language** – new properties and methods can be added runtime.
- JS supports object inheritance using **prototypes** and **mixins** (adding dynamically new properties and methods).
- **Prototypes** are **objects** (which also can have their prototypes) → **inheritance = traversing prototype chain**
- Main resource: **Introduction to OO JS YouTube video**
<https://www.youtube.com/watch?v=PMfcsYzj-9M>

JavaScript Features

- Supports **for ... in** operator for iterating object's properties, including inherited ones from the prototype chain.
- Provides a number of predefined datatypes such as: **Object, Number, String, Array, Function, Date** etc.
- **Dynamically typed** – variables are universal containers, no variable type declaration.
- Allows dynamic script evaluation, parsing and execution using **eval()** – **discouraged as a bad practice**.

Datatypes in JavaScript

- Primitive datatypes:
 - **boolean** – values **true** и **false**
 - **number** – floating point numbers (no real integers in JS)
 - **string** – strings (no **char** type → string of 1 character)
- Abstract datatypes:
 - **Object** – predefined, used as default prototype for other objects (defines some common properties and methods for all objects: **constructor**, **prototype**; methods: **toString()**, **valueOf()**, **hasOwnProperty()**, **propertyIsEnumerable()**, **isPrototypeOf()**;))
 - **Array** – array of data (really dictionary type, **resizable**)
 - **Function** – function or object method (defines some common properties: **length**, **arguments**, **caller**, **callee**, **prototype**)

Datatypes in JavaScript

- Special datatypes:
 - **null** – special values of **object type** that does not point anywhere
 - **undefined** – a value of variable or argument that have not been initialized
 - **NaN** – Not-a-Number – when the arithmetic operation should return numeric value, but result is not valid number
 - **Infinity** – special numeric value designating infinity ∞
- Operator **typeof**
Example: **typeof myObject.toString** //-->'function'

Functional JavaScript

- Functional language – functions are “first class citizens”
- Functions can have own **properties and methods**, can be assigned to variables, pass as arguments and returned as a result of other function's execution.
- Can be called by reference using operator **()**.
- Functions can have embedded inner functions at arbitrary depth
- All arguments and variables of outer function are accessible to inner functions – even after call of outer function completes
- Outer function = **enclosing context (Scope)** for inner functions → **Closure**

Closures

Example:

```
function countWithClosure() {  
    var count = 0;  
    return function() {  
        return count++;  
    }  
}
```

var count = countWithClosure(); <-- Function call – returns inner function which keeps reference to **count** variable from the outer scope

```
console.log( count() );    <-- Prints 0;  
console.log( count() );    <-- Prints 1;  
console.log( count() );    <-- Prints 2;
```

Default Values & RegEx

- Functions can be called with different number of arguments. It is possible to define default values – Example:

```
function Polygon(strokeColor, fillColor) {  
    this.strokeColor = strokeColor || "#000000";  
    this.fillColor = fillColor || "#ff0000";  
    this.points = [];  
    for (i=2; i < arguments.length; i++) {  
        this.points[i] = arguments[i];  
    }  
}
```

- Regular expressions – Example: `/a*/.match(str)`

Object Literals. Using **this**

- Object literals – example:


```
var point1 = { x: 50, y: 100 }
```

```
var rectangle1 = { x: 200, y: 100, width: 300, height: 200 }
```

- Using **this** calling a function /D. Crockford/ - „Method Call“:

```
var scene1 = {  
  name: 'Scene 1',  
  numElements: 5,  
  toString: function() {  
    return "Name: " + this.name + ", Elements: " + this['numElements'] }  
}  
console.log(scene1.toString()) // --> 'Name: Scene 1, Elements: 5'
```

Refers to object and allows access to its properties and methods



Accessing **this** in Inner Functions

- Using **this** calling a function /D. Crockford/ - „Function Call“:

```
var scene1 = {  
  ...  
  log: function(str) {  
    var self = this;  
    var createMessage = function(message) {  
      return "Log for " + self.name + " („ + Date() + “): “  
        + message;  
    }  
    console.log( createMessage(str) );  
  }  
}
```

It's necessary to use additional variable,
because **this** points to global object (window)
undefined in strict mode

„Classical“ Inheritance, call() apply() & bind()

- Pattern „Calling a function using special method“
Function.prototype.apply(thisArg, [argsArray])
Function.prototype.call(thisArg[, arg1, arg2, ...])
Function.prototype.bind(thisArg[, arg1, arg2, ...])

```
function Point(x, y, color){  
    Shape.apply(this, [x, y, 1, 1, color, color]);  
}  
extend(Point, Shape);  
  
function extend(Child, Parent) {  
    Child.prototype = new Parent;  
    Child.prototype.constructor = Child;  
    Child.prototype.supper = Parent.prototype;  
}
```

„Classical“ Inheritance. Using call() & apply()

```
Point.prototype.toString = function() {  
    return "Point [" + this.supper.toString.call( this ) + "];"  
}
```

```
Point.prototype.draw = function(ctx) {  
    ctx.fillStyle = this.fillColor;  
    ctx.fillRect(this.x, this.y, 1, 1);  
}
```

```
point1 = new Point(200,150, „blue“);  
console.log(point1.toString() );
```

„Classical“ Inheritance. Using call() & apply()

```
Point.prototype.toString = function() {  
    return "Point [" + this.supper.toString.apply( this, [] ) + "];"  
}
```

```
Point.prototype.draw = function(ctx) {  
    ctx.fillStyle = this.fillColor;  
    ctx.fillRect(this.x, this.y, 1, 1);  
}
```

```
point1 = new Point(200,150, „blue“);  
console.log(point1.toString() );
```


EcmaScript 6 – ES 2015, Harmony

[<https://github.com/lukehoban/es6features>]

A lot of new features:

- arrows
- classes
- enhanced object literals
- template strings
- destructuring
- default + rest + spread
- let + const
- iterators + for..of
- Generators
- unicode
- Modules + module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- Promises
- math + number + string + array + object APIs
- binary and octal literals
- reflect api
- tail calls

ES6 Classes [<http://es6-features.org/>]

```
class Shape {  
  constructor (id, x, y) {  
    this.id = id  
    this.move(x, y)  
  }  
  move (x, y) {  
    this.x = x  
    this.y = y  
  }  
}
```

```
class Rectangle extends Shape {  
  constructor (id, x, y, width, height)  
  {  
    super(id, x, y)  
    this.width = width  
    this.height = height  
  }  
}  
  
class Circle extends Shape {  
  constructor (id, x, y, radius) {  
    super(id, x, y)  
    this.radius = radius  
  }  
}
```

Block Scope Vars: let [<http://es6-features.org/>]

```
for (let i = 0; i < a.length; i++) {  
  let x = a[i]  
  ...  
}
```

```
for (let i = 0; i < b.length; i++) {  
  let y = b[i]  
  ...  
}
```

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

```
callbacks[0]() === 0  
callbacks[1]() === 2  
callbacks[2]() === 4
```

ES6 Arrow Functions and this

- ECMAScript 6:

```
this.nums.forEach((v) => {  
  if (v % 5 === 0)  
    this.fives.push(v)  
})
```

- ECMAScript 5:

```
var self = this;  
this.nums.forEach(function (v) {  
  if (v % 5 === 0)  
    self.fives.push(v);  
});
```

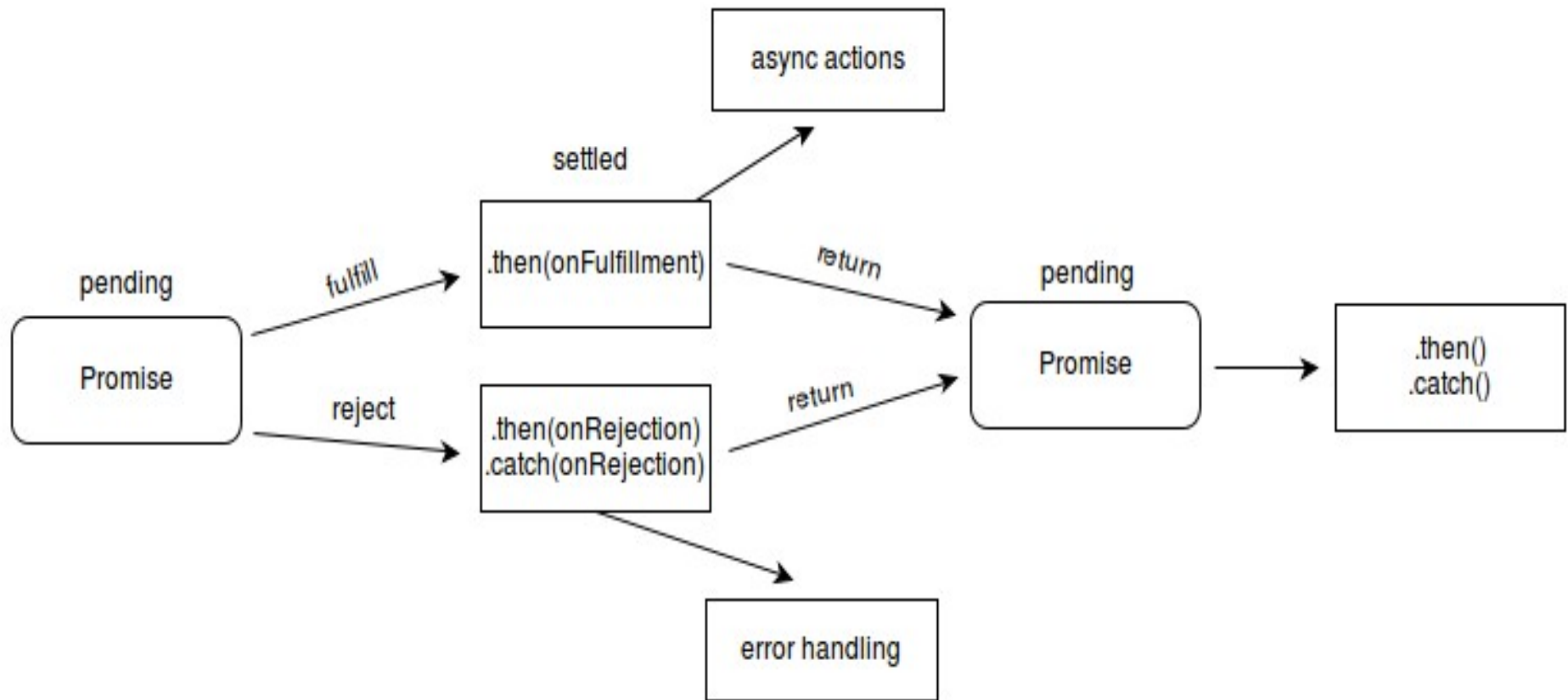
Array and Object Destructuring

```
let persons = [  
  { name: 'Michael Harrison',  
    parents: {  
      mother: 'Melinda Harrison',  
      father: 'Simon Harrison',  
    }, age: 35},  
  { name: 'Robert Moore',  
    parents: {  
      mother: 'Sheila Moore',  
      father: 'John Moore',  
    }, age: 25}];  
  
for (let {name: n, parents: { father: f }, age } of persons) {  
  console.log(`Name: ${n}, Father: ${f}, age: ${age}`);  
}
```

ES6 Promises [<http://es6-features.org/>]

```
function msgAfterTimeout (msg, who, timeout) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => resolve(`${msg} Hello ${who}!`), timeout)  
  })  
}  
  
msgAfterTimeout("", "Foo", 1000).then((msg) => {  
  console.log(`done after 1000ms:${msg}`);  
  return msgAfterTimeout(msg, "Bar", 2000);  
}).then((msg) => {  
  console.log(`done after 3000ms:${msg}`)  
})
```

ES6 Promises



Source:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Combining ES6 Promises

```
function fetchAsync (url, timeout, onData, onError) { ... }
fetchPromised = (url, timeout) => {
  return new Promise((resolve, reject) => {
    fetchAsync(url, timeout, resolve, reject)
  })
}
Promise.all([
  fetchPromised("http://backend/foo.txt", 500),
  fetchPromised("http://backend/bar.txt", 500)
]).then( (data) => {
  let [ foo, bar ] = data
  console.log(` success: foo=${foo} bar=${bar}`)
}).catch( (err) => {
  console.log(` error: ${err}`)
})
```

Combining ES6 Promises

```
function fetchAsync (url, timeout, onData, onError) { ... }
fetchPromised = (url, timeout) => {
  return new Promise((resolve, reject) => {
    fetchAsync(url, timeout, resolve, reject)
  })
}
Promise.all([
  fetchPromised("http://backend/foo.txt", 500),
  fetchPromised("http://backend/bar.txt", 500)
]).then( (data) => {
  let [ foo, bar ] = data
  console.log(` success: foo=${foo} bar=${bar}`)
}, (err) => {
  console.log(` error: ${err}`)
})
```

Async – Await – Try – Catch

```
async function init() {  
  try {  
    const userResult = await fetch("user.json");  
    const user = await userResult.json();  
    const gitResp = await fetch(  
      `http://api.github.com/users/${user.name}`);  
    const githubUser = await gitResp.json();  
    const img = document.createElement("img");  
    img.src = githubUser.avatar_url;  
    document.body.appendChild(img);  
    await new Promise((resolve, reject) => setTimeout(resolve, 6000));  
    img.remove();  
    console.log("Demo finished.");  
  } catch (err) {  
    console.log(err);  
  }  
}
```

JavaScript Module Systems - CommonJS

- math.js:

```
exports.add = function() {  
    var sum = 0, i = 0, args = arguments, len = args.length;  
    while (i < len) {  
        sum += args[i++];  
    }  
    return sum;  
};
```

- increment.js:

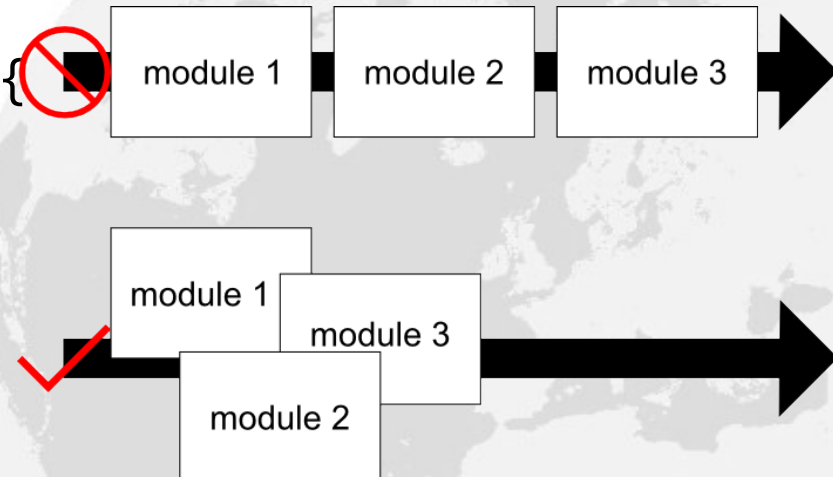
```
var add = require('./math').add;  
exports.increment = function(val) {  
    return add(val, 1);  
};
```

JavaScript Module Systems – AMD I

```
//Calling define with module ID, dependency array, and factory  
//function
```

```
define('myModule', ['dep1', 'dep2'], function (dep1, dep2) {  
    //Define the module value by returning a value.  
    return function () {};  
});
```

```
define(["alpha"], function (alpha) {  
    return {  
        verb: function(){  
            return alpha.verb() + 2;  
        }  
    };  
});
```



JavaScript Module Systems - AMD II

- Asynchronous module definition (AMD) – API for defining code modules and their dependencies, loading them asynchronously, on demand (lazy), dependencies managed, client-side

```
define("alpha", ["require", "exports", "beta"],  
    function(require, exports, beta) {  
        exports.verb = function() {  
            return beta.verb();  
            //OR  
            return require("beta").verb();  
        }  
    });  
  
define(function (require) {  
    require(['a', 'b'], function (a, b) { //use modules a and b  
    });  
});
```

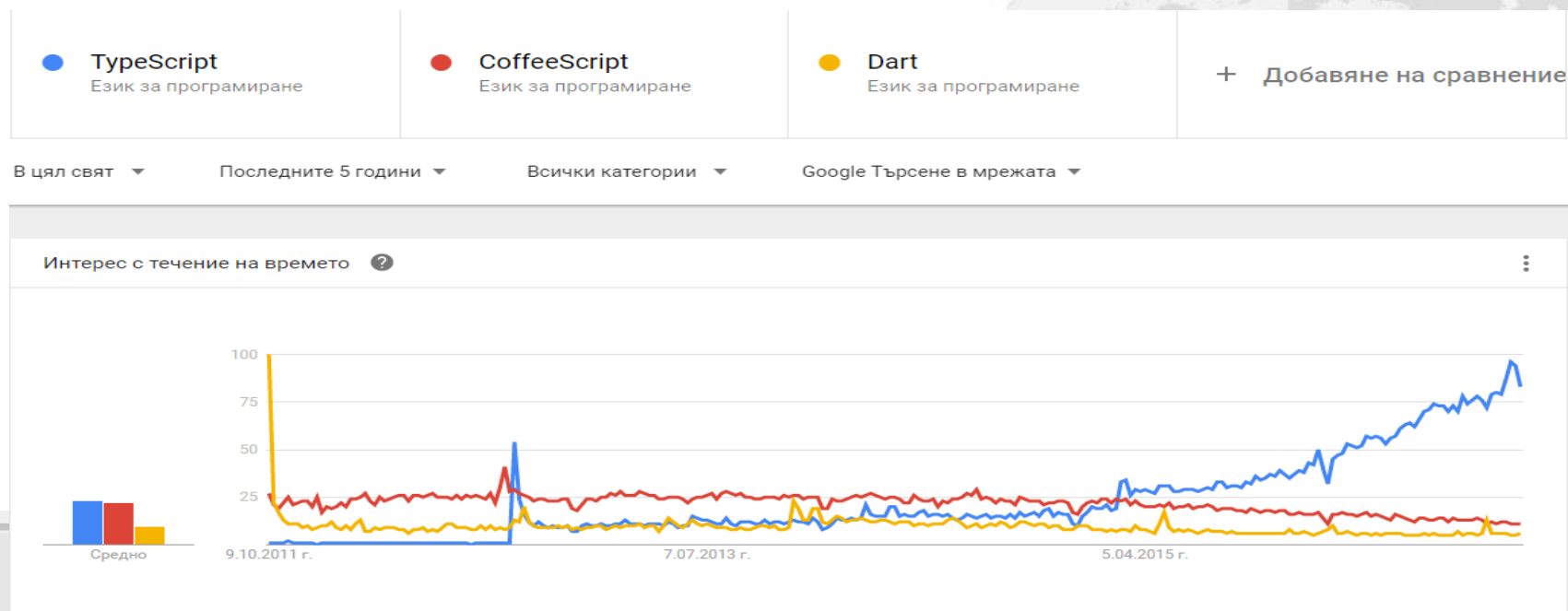
JavaScript Module Systems – ES6

- `// lib/math.js`
`export function sum (x, y) { return x + y }`
`export var pi = 3.141593`
- `// someApp.js`
`import * as math from "./lib/math"`
`console.log("2 π = " + math.sum(math.pi, math.pi))`
- `// otherApp.js`
`import { sum, pi } from "./lib/math"`
`console.log("2 π = " + sum(pi, pi))`
- `// default export from hello.js and import`
`export default () => (<div>Hello from React!</div>);`
`import Hello from "./hello";`

TypeScript

[<http://www.typescriptlang.org/>]

- Typescript → since October 2012, Anders Hejlsberg (lead architect of C# and creator of Delphi and Turbo Pascal)
- Targets large scale client-side and mobile applications by compile time type checking + @Decorators -> Microsoft, Google
- TypeScript is strictly superset of JavaScript, so any JS is valid TS



Source: Google Trends comparison

TypeScript Hello World I

- Installing Typescript: **npm install -g typescript**
- Create new directory: **md 02-ts-demo-lab**
- Create a new TS project using npm: **npm init**
- Write a simple TS function – file **greeter.ts** :

```
function greeter(person: string) {  
    return 'Hello, ' + person + ' from Typescript!';  
}  
  
const user = 'TypeScript User';  
document.body.innerHTML = greeter(user);
```

TypeScript Hello World II

- Compile TypeScript to JavaScript (ES5): **tsc greeter.ts**

- Include script in **index.html** :

```
<html>
<head>
  <title>ES6 Simple Demo 01</title>
</head>
<body>
  <script src="greeter.js"></script>
</body>
</html>
```

- And open it in web browser – thats all :)
- If you make changes - use watch flag: **tsc -w greeter.ts**

Configuring, Building and Deploying TypeScript Project

- Node package manager (**npm**) configuraton – **package.json**
- TypeScript configuration – **tsconfig.json**
- Configuring System.js module loader – **systemjs.config.js**
- Using external JS librarries – **@types** and npm packages
- TypeScript compiler options:
<http://www.typescriptlang.org/docs/handbook/compiler-options.html>
- Linting TypeScript code – **tslint.json**:
<https://palantir.github.io/tslint/rules/>
- Developing simple TS project – **Login Demo**

Visual Studio Code

The screenshot displays the Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, and Help. The title bar shows the active file: login-controller.ts - ts-demo-03-lab - Visual Studio Code. The left sidebar contains the Explorer, Search, and Extensions views. The Extensions view is active, showing a list of installed and available extensions. The 'Angular v2 TypeScript Snippets' extension by johnpapa is highlighted. The main editor area shows the details of this extension, including its description, version (2.0.3), and a preview of the snippets it provides. The bottom status bar shows the current file is login-controller.ts, the language is TypeScript, and there are 15 symbols and 321 errors/warnings.

login-controller.ts | Extension: Angular v2 TypeScript Snippets | user-repository.ts | index.ts

Search Extensions in Marketplace

Add jsdoc comments 0.0.8
Adds jsdoc @param and @return tags for s...
stevenc1

Angular 2 TypeScript Test Snippets 0.7.3
Angular 2 TypeScript test snippets (updated...
Marinho Brandao

Angular 2, 4 and upcoming latest TypeScript
Angular 2, 4 and upcoming latest Typescript...
Balram Chavan

Angular v2 TypeScript Snippets 2.0.3
Angular with TypeScript snippets. This exten...
johnpapa

Autolinting for Javascript 1.3.0
Automatically activate the correct Javascript...
t-sauer

Debugger for Chrome 2.7.1
Debug your JavaScript code in the Chrome ...
Microsoft

Document This 0.4.3
Automatically generates detailed JSDoc co...
Joel Day

ESLint 1.2.8
Integrates ESLint into VS Code.
Dirk Baeumer

exports autocomplete 0.4.6
autocompletes javascript module exports fr...
capaj

File Peek 1.0.1

Angular v2 TypeScript Snippets johnpapa.Angular2
johnpapa | 365100 | ★★★★★ | License
Angular with TypeScript snippets. This extension supports Angular v2 or greater.
Disable Uninstall

Details Contributions Changelog Dependencies

Angular TypeScript Snippets for VS Code

Now Updated for Angular 2.4 release

This extension for Visual Studio Code adds snippets for Angular for TypeScript and HTML.

villain.component.ts - angular-event-view-cli

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL**

1: cmd.exe

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\CourseAngular2\git\course-angular2\ts-demo-03-lab>

master* TSLint 0 2 [TypeScript Importer]: Symbols: 15 321

Start | Taskbar icons | System tray: EN, 11:57 PM, 3/25/2017

VS Code: Popular Angular/TS Plugins

- **TSLint** - linter for the TypeScript language, help fixing error in TS code. Must have when working with TS.
- **Angular Language Service** - This extension provides a rich editing experience for Angular templates, both inline and external templates.
- **Angular v5 TypeScript Snippets**
- **Angular 5 and TypeScript/HTML VS Code Snippets**
- **Angular + snippets - TypeScript, Html, NGRX, ...**
- **Angular Files** - allows quickly scaffold angular 2 file templates in VS Code project.


VS Code: Popular Angular/TS Plugins II

- **TypeScript Hero** - Sorts and organizes your imports according to convention and removes imports that are unused (Ctrl+Alt+o on Win/Linux or Ctrl+Opt+o on MacOS).
- **Path Intellisense** - VSCode has a very good auto import capability, but sometime you still need to import some files manually, and this extension helps a lot in these cases.
- **TypeScript Importer** - Automatically searches for TypeScript definitions in workspace files and provides all known symbols as completion item to allow code completion.
- **Debugger for Chrome** - allows to debug using chrome and add your breakpoints in VSCode.

Let's Install Some VSCode Plugins I


INSTALLED

43

**Add Angular Files** 2.0.1


22K ★4

Sebastian Baar

**Angular 2 TypeScript Emmet** 2.0.3


145K ★3

jakethashi

**Angular 5 and TypeScript/HTML VS Code Snippets** 1.0.19


288K ★5

Dan Wahlin

**Angular 5 Snippets - TypeScript, Html, Angular Material, ngRx, RxJS & Flex Layout** 5.2.23


2.1M ★5

Mikael Morlund

**Angular Essentials** 0.3.2


93K ★4.5

John Papa

**Angular Extension Pack** 1.0.2


82K ★4.5

Will 保哥

**Angular Files** 1.4.0


70K ★4.5

Alexander Ivanichev

**Angular Follow Selector** 1.1.1

4K ★5

Sander Ledegen










**Angular Language Service** 0.1.9

506K ★4.5

Let's Install Some VSCode Plugins II










43

INSTALLED

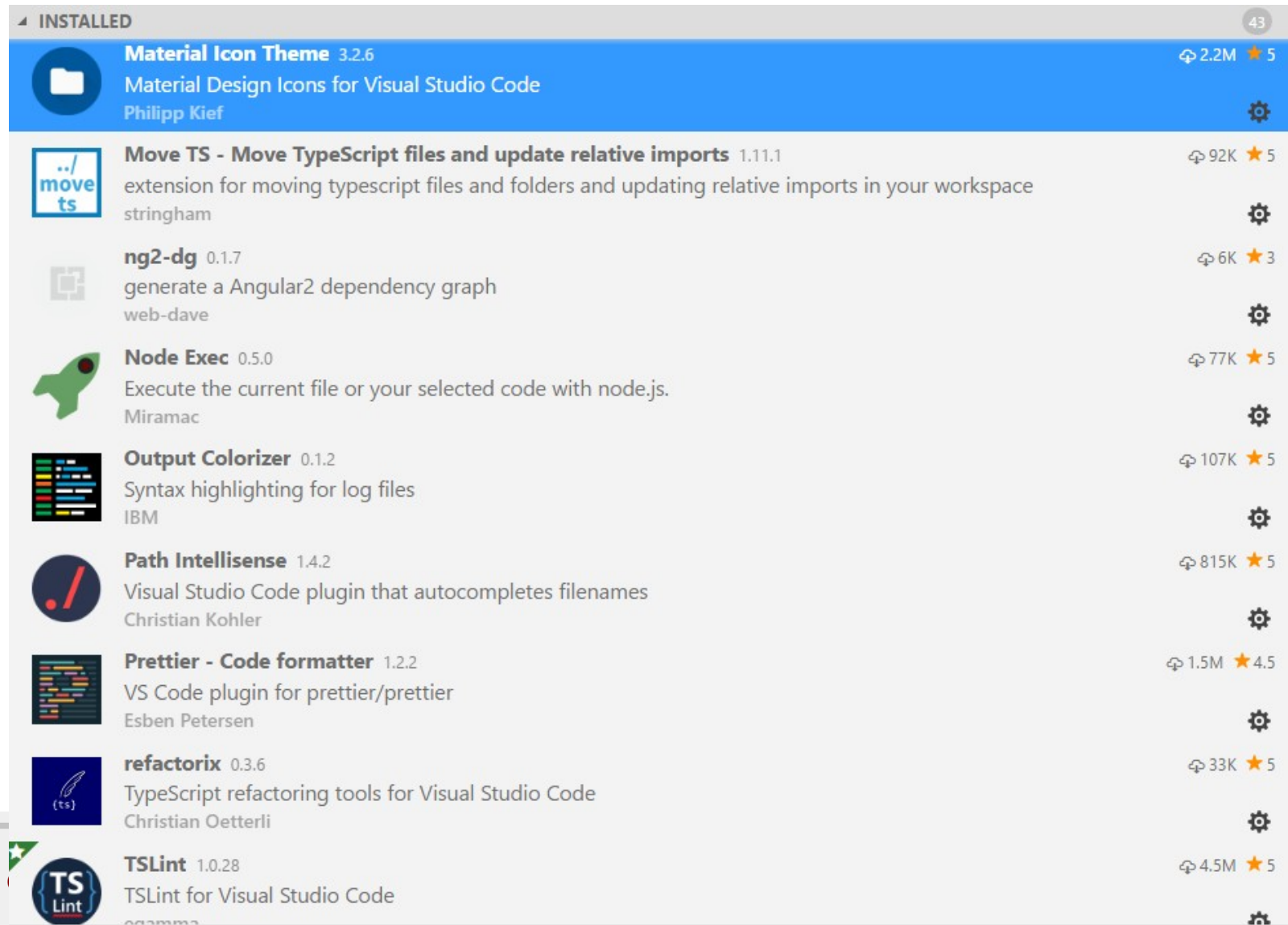
	Angular Material 2, Flex layout 1, Covalent 1 & Material icon snippets 0.13.0 Providers snippets Angular Material 2, Angular Flex layout 1, Teradata Covalent 1 & Material icons 1Ton Technologies	43K ★ 5
	Angular Productivity Pack 1.7.1 A collection of Angular extensions to boost productivity DevBoosts	14K ★ 5
	Angular UI Bootstrap Snippets 4.0.9 Angular UI Bootstrap Snippets for VS Code (HTML and JS) HerrHerrmann	64K ★ 5
	Angular v5 Snippets 2.14.0 Angular v5 snippets by John Papa John Papa	2.4M ★ 5
	angular2-inline 0.0.17 Visual Studio Code language extension for javascript/typescript files that use Angular2. Nate Wallace	124K ★ 4.5
	angular2-switcher 0.1.6 Easily navigate to component's files in angular2 project. infinity1207	103K ★ 5
	Auto Rename Tag 0.0.15 Auto rename paired HTML/XML tag Jun Han	541K ★ 4.5
	Beautify 1.3.0 Beautify code in place for VS Code HookyQR	2.3M ★ 4.5
	Better Comments 1.2.1 Improve your code commenting by annotating with alert, informational, TODOs, and more! Ammar Band	82K ★ 5

Let's Install Some VSCode Plugins III

INSTALLED 43

	CSS Peek 2.1.0 Allow peeking to css ID and class strings as definitions from html files to respective CSS. Allows peek and goto definition. Pranay Prakash	162K 4	⚙️
	Debugger for Chrome 4.2.1 Debug your JavaScript code in the Chrome browser, or any other target that supports the Chrome Debugger protocol. Microsoft	6.6M 4.5	⚙️
	Document This 0.6.0 Automatically generates detailed JSDoc comments in TypeScript and JavaScript files. Joel Day	595K 4	⚙️
	EditorConfig for VS Code 0.12.1 EditorConfig Support for Visual Studio Code EditorConfig	1.2M 5	⚙️
	ESLint 1.4.7 Integrates ESLint into VS Code. Dirk Baeumer	6.1M 4.5	⚙️
	Git History 0.4.0 View git log, file history, compare branches or commits Don Jayamanne	2.3M 4.5	⚙️
	HTML SCSS Support 0.0.42 SCSS support for HTML documents P-de-Jong	40K 4	⚙️
	JavaScript (ES6) code snippets 1.5.0 Code snippets for JavaScript in ES6 syntax charalampos karypidis	1.1M 4.5	⚙️
	JSON to TS 1.5.4 Convert JSON object to typescript interfaces MariusAlchimavicius	69K 4.5	⚙️

Let's Install Some VSCode Plugins IV








The screenshot displays the 'INSTALLED' tab in the VS Code Extensions sidebar. It lists 43 installed extensions. The first extension, 'Material Icon Theme', is highlighted in blue. The list includes various plugins for themes, file management, dependency graphs, code execution, log file formatting, file name completion, code formatting, TypeScript refactoring, and linting.

Extension Name	Version	Author	Downloads	Rating
Material Icon Theme	3.2.6	Philipp Kief	2.2M	5
Move TS - Move TypeScript files and update relative imports	1.11.1	stringham	92K	5
ng2-dg	0.1.7	web-dave	6K	3
Node Exec	0.5.0	Miramac	77K	5
Output Colorizer	0.1.2	IBM	107K	5
Path Intellisense	1.4.2	Christian Kohler	815K	5
Prettier - Code formatter	1.2.2	Esben Petersen	1.5M	4.5
refactorix	0.3.6	Christian Oetterli	33K	5
TSLint	1.0.28	egamma	4.5M	5

Let's Install Some VSCode Plugins V

43 INSTALLED

-  **TypeScript Hero** 2.3.1
Additional tooling for the typescript language
Christoph Bühler
569K ⭐ 4
-  **TypeScript Importer** 1.2.14
Automatically searches for TypeScript definitions in workspace files and provides all known symbols as completion item to..
pmneo
91K ⭐ 4.5
-  **TypeScript Toolbox** 0.4.2
Add and Optimize Imports, Generate Getters / Setters and Constructors
DSKWRK
109K ⭐ 4
-  **VSCode simpler Icons with Angular** 1.5.2
A fork from vscode-great-icons. With fewer different icons and some angular icons. Mainly created for personal use.
davidbabel
69K ⭐ 5
-  **vscode-icons** 7.22.0
Icons for Visual Studio Code
Roberto Huertas
6M ⭐ 5

Introduction to TypeScript I

- Functions, interfaces, classes and constructors.
- Common types – Boolean, Number, String, Array, Tuple, Enum, Any, Void, Null, Undefined.
- --strictNullChecks flag
- Type assertions: `let length: number = (<string> data).length;`
`let length: number = (data as string).length;`
- **Duck typing** = structural similarity based typing
- Declaring variables – **let**, **const** and **var**. Scopes, variable capturing, Immediately Invoked Function Expressions (IIFE), closures and **let**.

Declaring Contracts using Interfaces I

```
export interface User {  
  id: number;  
  firstName: string;  
  lastName: string;  
  email: string;  
  password: string;  
  contact?: Contact; ← Optional properties  
  roles: Role[];  
  getSalutation(): string; ← Required methods  
}
```


Declaring Contracts using Interfaces II

```
import { User } from './users';
export interface UserRepository {
  addUser(user: User): void;
  editUser(user: User): void;
  ...
  findAllUsers(): User[];
}
export class DemoUserRepository implements UserRepository {
  private users = new Map<number, User>();
  public addUser(user: User): void {
    if (this.findUserByEmail(user.email)) {
      throw `User with email ${user.email} exists.`;
    }
    user.id = this.getNextId();
    this.users.set(user.id, user);
  }
  ...
}
```

Declaring Contracts using Interfaces III

- Properties, optional properties and readonly properties:

```
interface UserRepository {  
    readonly size: number;  
    addUser: (user: User) => void;  
}
```

- Function types:

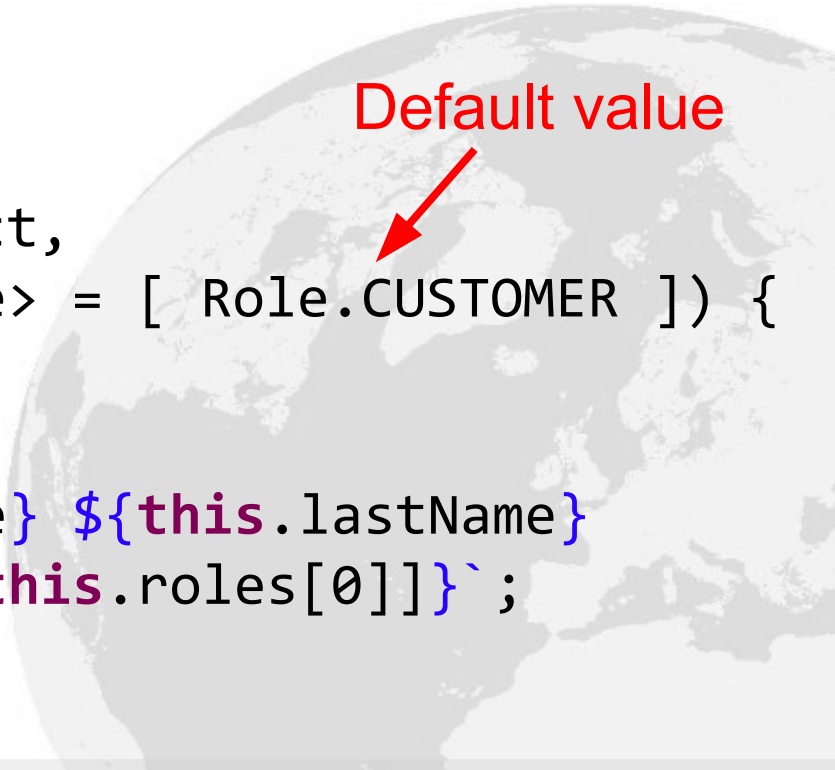
```
interface RoleFinder {  
    (user: User) : Role[];  
}
```

- Array (indexable) types. Dictionary pattern:


```
interface EmailUserMap {  
    [key: string]: User;  
}
```

Class Types

```
export class Customer implements User {  
    public id: number; // set automatically by repository  
    constructor(public firstName: string,  
        public lastName: string,  
        public email: string,  
        public password: string,  
        public contacts?: Contact,  
        public roles: Array<Role> = [ Role.CUSTOMER ]) {  
    }  
    public get salutation() {  
        return `${this.firstName} ${this.lastName}  
            in role ${Role[this.roles[0]]}`;  
    }  
}
```



Default value



Extension of Interfaces. Hybrid Types.

```
export interface Person {  
    id: number;  
    firstName: string;  
    lastName: string;  
    email: string;  
    contact?: Contact;  
}
```

```
export interface User extends Person {  
    password: string;  
    roles: Role[];  
    getSalutation(): string;  
}
```

Classes

- Constructors, constructor arguments as properties
- Public/private/protected properties
- Get and set accessors

```
export interface User extends Person{  
    password: string;  
    roles: Role[];  
    readonly salutation: string;  
} ...  
public get salutation() {  
    return `${this.firstName} ${this.lastName}`;  
}
```

- Static and instance sides of a class. Abstract classes

Functions and Function Types

- Optional, default and **rest (...)** parameters

```
export class Person {  
    public restNames: string[];  
    constructor(public firstName: string, ...restNames: string[]) {  
        this.restNames = restNames;  
    }  
    public get salutation() {  
        let salutation = this.firstName;  
        for (let name of this.restNames) {  
            salutation += ' ' + name;  
        }  
        return salutation;  
    }  
}  
console.log(new Person('Ivan', 'Donchev', 'Petrov').salutation);
```

Function Lambdas (\Rightarrow) and Use of **this**

```
export class LoginComponent {  
  constructor(private jqElem: string, private loginC: LoginController){  
    const keyboardEventHandler = (event: JQueryKeyEventObject) => {  
      if (event.keyCode === 13) {  
        loginEventHandler();  
      }  
    };  
  
    const loginEventHandler = () => {  
      this.loginC.login(usernameInput.val(), passwordInput.val())  
        .then(() => {  
          this.showCurrentUser();  
        }).catch(err => {  
          this.showError(err);  
        });  
      return false;  
    };  
    ...  
  }  
}
```


Using Lambdas

```
const usernameInputElem =  
  $("<input id='username' type='email'>")  
    .addClass('form-control')  
    .bind('keypress', keyboardEventHandler);
```

Type Guards & Method Overloading

```
export class DemoLoginController implements LoginController {
  public login(email: string, password: string): Promise<User>;
  public login(user: User): Promise<User>;
  public login(principal: User | string, credentials?: string)
    : Promise<User> {
    let email: string, password: string;
    if (typeof principal === 'string') {
      email = principal;
      password = credentials;
    } else {
      email = principal.email;
      password = principal.password;
    }
    let promise = new Promise<User>( (resolve, reject) => { ... });
    return promise;
  }
}
```

Using Enums

- Defining enumeration:

```
enum Role {  
    ADMIN = 1, CUSTOMER  
}
```

- In generated code an enum is compiled into an object that stores both forward (name -> value) and reverse (value -> name) mappings.
- Getting enumerated name by value:

```
public get salutation() {  
    return `${this.name} in role ${Role[this.roles[0]]}`;  
}
```

JavaScript Module Systems – ES6

- `// lib/math.js`
`export function sum (x, y) { return x + y }`
`export var pi = 3.141593`
- `// someApp.js`
`import * as math from "lib/math"`
`console.log("2 π = " + math.sum(math.pi, math.pi))`
- `// otherApp.js`
`import { sum, pi } from "lib/math"`
`console.log("2 π = " + sum(pi, pi))`

Modules in TypeScript

- **Namespaces** and **modules** – former internal and external modules – prefer **namespace X { ... }** instead **module X { ... }**
- ES6 modules (preferred) – using **export** and **import**:

```
export interface Person {...}
export interface User extends Person {...}
export interface Contact {...}
export enum Role { ADMIN, CUSTOMER }
export class Customer implements User {...}
import {User} from './users';
import {resolvePromiseAfterTimeout} from './utilities';
import $ from 'jquery';
import * as repository from './user-repository'; //default import
import './my-module.js'; //import a module for side-effects only
let module = await import('/modules/my-module.js'); // dynamic
```

Interoperability with External JS Libraries

- Ambient type declarations and ambient modules (***.d.ts**) – **typings, @types** – Example **node.d.ts** (simplified):

```
declare module "url" {  
    export interface Url {  
        protocol?: string;  
        hostname?: string;  
        pathname?: string;  
    }  
    export function parse(urlStr: string, parseQueryString?,  
        slashesDenoteHost?): Url;  
}  
/// <reference path="node.d.ts"/>  
import * as URL from "url";  
let myUrl = URL.parse("https://github.com/iproduct");
```

Generic Type Parameters I

- Writing generic functions, interfaces and classes – Examples:

```
interface Repository {  
    findById<T>(id: number): T;  
    findAll<T>(): Array<T>;  
}
```

//OR

```
interface Repository<T> {  
    findById: (id: number) => T;  
    findAll(): Array<T>;  
}
```


Generic Type Parameters II

```
export class RepositoryImpl<T> implements Repository<T> {  
    private data = new Map<number, T>();  
    public findById(id: number): T {  
        return this.data.get(id);  
    }  
    public findAll(): T[] {  
        let results: T[] = [];  
        this.data.forEach(item => results.push(item));  
        return results;  
    }  
}
```

- Bounded generics
- Generic constructors

Advanced TypeScript Topics

- Advanced types
- Type Guards and Differentiating Types
- Type Aliases
- Symbols
- Declaration merging
- Decorators
- Using mixins in TypeScript

Webpack Builder & Bundler Tool

webpack - Mozilla Firefox

File Edit View History Bookmarks Tools Help

we X Home Media Lib AdWo AdWo M [ANS] P How t P Mar 2 P Green HTML Web Softw Monit 25 Googl T CALL P Openi BTA Flying FX Impre Graphi f > + v

https://webpack.js.org

Sponsor webpack and get apparel from the [official shop](#) or get stickers [here](#)! All proceeds go to our [open collective](#)!

webpack

DOCUMENTATION CONTRIBUTE VOTE BLOG

bundle your scripts

MODULES WITH DEPENDENCIES

STATIC ASSETS

Windows taskbar: 10:08 PM 4/1/2018

Creating New Project: NPM + WebPack

[<https://angular.io/guide/webpack>]

```
mkdir my-project
cd my-project
npm init
npm install webpack webpack-dev-server --save-dev
touch index.html src/index.js webpack.config.js
npm install awesome-typescript-loader angular-router-loader
  angular2-template-loader --save-dev
npm install css-loader style-loader css-to-string-loader --save-dev
npm install file-loader url-loader html-loader --save-dev
npm install extract-text-webpack-plugin html-webpack-plugin --save-dev
```

In package.json:

```
"scripts": {
  "start": "webpack-dev-server --inline --hot",
  "watch": "webpack --watch",
  "build": "webpack -p"
},
```

Simple webpack.config.js I

```
const path = require('path');
```

```
module.exports = {  
  context: path.resolve(__dirname, 'src'),  
  entry: './index.js',  
  output: {  
    path: path.resolve(__dirname, 'dist'),  
    filename: 'bundle.js'  
  },  
  ...  
}
```

Simple webpack.config.js II

```
module: {  
  rules: [{  
    test: /\.ts$/,  
    loaders: ['awesome-typescript-loader',  
      'angular-router-loader', 'angular2-template-loader']  
  }, {  
    test: /\.html$/,  
    loader: 'html-loader'  
  }, ...  
  {  
    test: /\.css$/,  
    include: helpers.root('src', 'app'),  
    loader: ['css-to-string-loader', 'css-loader']  
  }]  
}  
...
```

Webpack Loaders and Plugins

- Loaders are **transformations** (functions running in node.js) that are applied on a resource file of your app
- You can use loaders to load **ES6/7** or **TypeScript**
- Loaders can be chained in a pipeline to the resource. The final loader is expected to return **JavaScript**
- Loaders can be **synchronous** or **asynchronous**
- Loaders accept **query parameters** – loader configuration
- Loaders can be **bound to extensions / RegExps**
- Loaders can be published / installed through **npm**
- **Plugins** – more universal than loaders, provide **more features**

Webpack Loaders

[<https://webpack.js.org/loaders/>]

- **Ts-loader, awesome-typescript-loader** - TypeScript => ES 5 or 6
- **babel-loader** - turns ES6 code into vanilla ES5 using Babel
- **file-loader** - emits the file into the output folder and returns the url
- **url-loader** - like file loader, but returns Data Url if file size <= limit
- **extract-loader** - prepares HTML and CSS modules to be extracted into separate files (alt. to ExtractTextWebpackPlugin)
- **html-loader** - exports HTML as string, requiring static resources
- **style-loader** - adds exports of a module as style to DOM
- **css-loader** - loads css file resolving imports and returns css code
- **sass-loader** - loads and compiles a SASS/SCSS file
- **postcss-loader** - loads and transforms a CSS file using PostCSS
- **raw-loader** - lets you import files as a string

Webpack Main Plugins

- [ExtractTextWebpackPlugin](#) - extracts CSS from your bundles into a separate file (e.g. app.bundle.css) → [mini-css-extract-plugin](#)
- [CompressionWebpackPlugin](#) - prepare compressed versions of assets to serve them with Content-Encoding
- [I18nWebpackPlugin](#) - adds i18n support to your bundles
- [HtmlWebpackPlugin](#) - simplifies creation of HTML files (index.html) to serve your bundles
- [ProvidePlugin](#) - automatically loads modules, whenever used
- [UglifyJsPlugin](#) – tree transformer and compressor which reduces the code size by applying various optimizations
- ~~[CommonsChunkPlugin](#)~~ - generates chunks of common modules shared between entry points and splits them to separate bundles

Resources

- Official Webpack repo @ GitHub:
<https://github.com/webpack/webpack>
- Webpack: An Introduction (Angular website):
<https://v5.angular.io/guide/webpack>
- TypeScript Quick start –
<http://www.typescriptlang.org/docs/tutorial.html>
- TypeScript Handbook –
<http://www.typescriptlang.org/docs/handbook/basic-types.html>



Thanks for Your Attention!

Questions?