# Angular2

Typescript

# What is TypeScript

- Programming language that compiles to JavaScript
- Open source and maintained by Microsoft
- Superset of JavaScript
- Optional static type and type checking
- ES6 Support
- Browsers can’t execute TypeScript files
- Typescript comes with a compiler
- Better IDE completion than JS

# Installing TypeScript

- we use npm to install typescript
- Need to init npm in a local folder: **npm init --yes**
- To install typescript
  - **npm install typescript --save-dev**
- you can verify typescript is installed by typing:
  - **tsc -v**
- For visual studio users you can start a new Blank Node.js Web Application and run the commands above from inside visual studio
-
-

# Hello World

- Typescript files have a **.ts** extension
- create **hello.ts** and using the **console.log** we will print **hello world**
- Browsers don’t understand typescript files and the file need to be turned to JS
- we use the Typescript compiler to turn the typescript files to javascript files
- Basic usage of the compiler
  - **tsc hello.ts**
  - this will create **hello.js** from the typescript files we created
- You can view additional options of the compiler by typing: **tsc --help**
- some interesting options:
  - **target, outDir, sourcemap, watch, module**

# TypeScript configuration - tsconfig.json

- configuration file for typescript
- usually located at the root dir of the typescript project
- some of the compiler options we seen earlier can be specified in this file
- compiler will look in the tsconfig options and compile according to that file
- **tsc --init** will output that config file
- compiling with no input file will search for tsconfig.json file in the current dir and it’s parents
- You can add a **--project (-p)** flag to specify the tsconfig.json directory
- the compilation will be according to the files listed in the tsconfig or if not listed in the tsconfig directory and all sub directory will look for **ts** files
- When input files are specified in the compiler cli then tsconfig is ignored
- lets create a default tsconfig to examine popular options.

# Debugging typescript files

- browsers can’t run typescript files so they will run our compiled JS files
- We want to debug our code with our written code and not generated code
- source map is a file that maps from the transformed source to the original source
- using the source map the browser can reconstruct the original source and place that source in the debugger
- the browser will know about the source map by a special comment at the end of the js
- there is an option that you can place in **tsconfig** that will create the source map: **compilerOptions.sourceMap = true**

# Variable Declaration - var

- Variables in typescript are defined like JS with: **var**
- Syntax: **var <variableName> = <assignment>;**
- assignment is optional
- **variableName** should be camelcased
- variable type can change dynamic language/loosly typed
- example:

```
var myString = ‘hello world’

myString = 10;
```

- What is the scope of var?

# Variable Declaration - const, let

- Syntax:

```
const <variableName> = <assignment is a must>;
let <variableName > = <assignment is optional>
```

- const has a single assignment
- let can have multiple assignment
- Is single assignment mean immutable?
- The scope of let and const is inside the block
- What’s the result of the previous example when changing var to let ?

# Static Types, Type Checking

- With TypeScript you can optionally specify the type of variable
- The syntax is:
  - **const/let/var variableName: <type> = assignmentIsOptionalUnlessConst**
- When compiling the file typescript will check that the type is matching
- Typescript will also try to guess the type (**Inferred type**)

# Basic Types - string

- **let myString: string = ‘hello world’;**
- Define a string: “”, ‘’, ``
- `` backticks are used for multiple lines
- `` with backticks you can inject javascript variables by using ${}
- you can concat strings with the **+**
- string is an array of characters so you can access a character like array syntax: **myStr[i]** (you can’t change the value, string are immutables)
- you can iterate on a string like array

- Some common functions: **indexOf, substr, split**

# Basic Types - Numbers

- single number type that represents: float, positive, negative numbers
- Operators: +, -, *, /, %, **, ++, --
- **toString** will convert number to string
- **parseInt, parseFloat** will convert from string to number if fails will return **NaN**
- numbers are immutable
- number constants: **NaN, Infinity, -Infinity**

# Booleans

- **const myBoolean: boolean = true;**
- true, false
- common tricks with boolean:
  - if (<var>) { … } // ‘’, 0, null, undefined, NaN are false
  - const myVar = expressionIfTrue || -1
- booleans are immutable
- logical operators: !, ==, ===, ||, &&, !=, !==

# Basic Types - Miscellaneous

- undefined
- null
  - null and undefined are subtype of everything unless --**strictNullChecks true**
- NaN
- Infinity
- -Infinity
- void
- any
- Object

# Type assertion - casting

**Example:**

```
var person : any =  ‘yariv kayz’;

var nameLength : number = (<string>person).length;

var nameLength2 : number = (person as string).length;
```

# Advanced Types - Arrays

Syntax:

- **const myNumArray: number[] = [];**
- **const myStringArray: Array<string> = []**

- TypeScript will check when you push to the array that the type match
- if you want to support different types you can: **const myAnyArray: any[] = [];**
- common methods: **forEach, push, pop, splice,**
- common properties: **length**

# Advanced Types - Object/Dictionary

- syntax:
  - **const dict: {[key: string]: any} = {<string key>: <value>, <string key2>: <value2>}**
- access values: **dict.key1** or **dict[‘key1’]**
- add value: **dict[‘newkey’] = <new value>**
- get an array of all the keys: **Object.keys(dict)**
- delete a key: **delete dict[‘newkey’]**
- is key in object? **dict.hasOwnProperty(‘newkey’)**

# for… of, for… in

- What does each loop is used for?
- is one of them dangerous and if so how?

# function - define a function

- Function can return type
  - **function(x: number, y: number): number { return  x + y; }**
- You can define a variable to accept a function
  - **let pokeFunc : (message : string) => void = function(msg){console.log(msg);}**
- The compiler will check if you call the function with the correct number of arguments

# function - arguments

- arguments can get type
- compiler will check the types that are passed to functions
- you can access the function arguments from: **arguments** array
- you can pass default value to arguments
- default arguments don’t have to be the last ones
- You can supply an optional param by adding **?**
  - **function(x: number, y?: number): number { return  x + y; }**
  - The optional params must be last
-

# function - quiz

- parameters to function are passed by reference or by value?
- can you name 3 ways to call a function? can you tell the difference between them?

# this

- this behaves differently in JS then in other languages
- By default **this === window**
- when a function is called this is equal to window
- when a function has ‘use strict’ this is equal to undefined
- by default TypeScript won’t ‘use strict’ you can set in **tsconfig:**
  - **compilerOptions.alwaysStrict = true**
- this will be determined at call time
- when a function is part of an object this will be the object
- when a function is called with the **new** then **this** will be the new object of the function (good for dealing with classes)
- you can use **bind** to set what **this** will be

# Lambda Functions

- syntax
  - (arg1, arg2) => { … }
  - arg1 => { … }
  - (arg1, arg2) => 3 // return 3
  - arg1 => 3
- doesn’t have a **this**

# Prototype

- JavaScript doesn’t have a subclass and inheritance like traditional languages
- JavaScript uses prototype to achieve this
- The base prototype is: **Object.prototype** nearly all object are instances of **Object**
- Some of  the inherited methods: **toString, hasOwnProperty, create, getPrototypeOf, constructor**
- **Array** and **Function** has prototype as well which inherits from **Object.prototype**
- when searching for a property it will start from the nearest prototype and then search in the next one and next one (prototype chaining)
- the next prototype is saved in the **__proto__**
- we can use prototype to create classes and inheritance
- We can take advantage of prototype chaining and override methods in the chain

# Class

- Class is a syntax sugar for creating a class and inheritance like common languages and not by using prototype
- The feature was added in ES6
- you can define **constructor** in the class
- In the constructor arguments you can specify if an argument will be saved as **private, public, protected**
- inheritance is done with the **extend** keyword
- you can call base function by using **super** (in constructor it has to be the first statement)
- you can define static methods with the keyword **static**
- you can define getters and setters
- abstract class

# Interfaces

- Syntax:
  - **interface IInterfaceName {...}**
- can include optional properties
- can define a function
- can define methods that a class needs to implement

# enum

- Giving a friendly name to numeric values
- Syntax:
  - **enum Color {Red, Green, Blue};**
  - **var c : Color = Color.Red;**
- By default the numbering starts from zero
- You can change the by specifying the first item
- After you specify an item the rest will be start incrementing from that value
- You can specify the values of all the items
  - **enum Color {Red = 1, Green = 4, Blue = -1}**
- You can also get the string from the key
  - **var colorName: string = Color[2];**

# Generic Type

- generic type can be added to **function, interface, class**
- With generic type your object behaviour changes according to the type sent
- you can restrict the generic type by using **extends**

# Decorators

- used to annotate or modify class or class members
- currently at **stage 2** (still experimental and syntax might change)
- to enable in **tsconfig** specify **compilerOptions.experimentalDecorators=true**
- decorator is a function that gets called with the decorated object
- decorators are more common to use with a decorator factory which allows to add configuration to the decorator
- decorators can be attached to: **class, method, accessor, property**

# Todo Rest Server

- our rest server is located at this url: **https://nztodo.herokuapp.com**
- The server is connected to a database with a single table called task
- the task table api is in this path: **/api/task/**
- The server returns a **json** response

# Task JSON

- A single task json looks this:

**{"id":8529,"title":"mytitle","description":"mydescription","group":"mygroup","when":"2016-12-12T21:20:00Z"}**

- **id** is the primary key and automatically created by the server
- **when** is an **ISOString** representing date time

# CORS

- stands for **Cross-Origin Resource Sharing**
- As a security measure browsers restrict cross-origin HTTP requests initiated from within scripts
- using CORS spec we can do cross domain communication between browser and server
- CORS are used with HTTP headers
- CORS headers has **Access-Control-*** prefix
- **Access-Control-Allow-Origin** - is required in the response from the server
- Certain Requests for the server are considered simple and are sent directly to the server
- some requests like **PUT, DELETE** the browser will automatically send a preflight request

# GET all tasks from server

**host: https://nztodo.herokuapp.com**

**path: /api/task/?format=json**

**method: GET**

- fetch will work with promise
- fetch will return promise even on bad response
-

# Get a single task

**host: https://nztodo.herokuapp.com**

**path: /api/task/:id/?format=json**

**method: GET**

# Insert new task

**host: https://nztodo.herokuapp.com**

**path: /api/task/**

**method: POST**

**request body: {title: …, description: …, when: …, group: ...}**

# Delete

**host: https://nztodo.herokuapp.com**

**path: /api/task/:id/**

**method: DELETE**

# UPDATE

**host: https://nztodo.herokuapp.com**

**path: /api/task/:id/**

**method: PUT**

**request body: {title: …, description: …, when: …, group: ...}**

# Modules

- With modules we can split our project to multiple files
- you can tell the compiler to concat all the files to a single file with the option: **compilerOptions.outFile**
  - this will work with module system **amd** or **system**
- concat to a single file is not recommended
- for now to use modules we need to include the entry point file in the index with type **module**

# Modules - export

- using export you can expose **function, class, constant** that can be imported from other module
- you can use export to chain export from other files (good for barrel files)
- you can use export default then import name can change
- if using regular export then name is important

# Modules - import

- you can import exported **functions, const, class**
- exported default items can be imported with any name
- export without default name should persist in import as well
- you can use **import * as name from …** to import everything in a module
- you can change the name of the import with alias
- import can be relative or non relative
  - relative will start with / ./ ../ - use it to point to your own modules, relative to the importing file
  - non relative: **import * as $ from ‘jquery’;**
  - non relative used for external dependency
  - **tsconfig compilerOptions.moduleResolution** will determine how non relative will be searched
- import without specifying what will just run the file

# Teaching the compiler

- some api’s are not recognized by the compiler
  - example **fetch**
- we still want to use them and we still want the compiler to check that we are using them correctly
- in the **tsconfig** you can add **lib** array with string of additional packages that the compiler should know about
- you can use the **declare** to make the compiler aware of something global
  - **declare var fetch : any;**
- you can use definetly typed **@types** to download to the compiler interfaces for popular packages.

# Assignment

- create a task class with the fields you are getting from the server
- the class gets in the constructor a json object similar to what the server returns from that modifies the fields
- try and fetch the tasks from the server
- try to create instances of the class for every json the server returns
- print the tasks title in the DOM using jquery
-