TypeScript

Agenda

- TypeScript Overview
- Types & Arrays
- ECMA Script 6+ Essentials
- Functions, Classes, Interfaces
- Generics, Modules, Decorators
- Consuming Services
- Bundling TS using Webpack

TypeScript Overview

What is TypeScript

- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript
- Allows use of classes and other features in browsers that do not support ECMA Script 6
- Compiled by the TypeScript compiler from *.ts to *.js by „tsc.exe“
- Language spec on http://www.typescriptlang.org/
- Co-authored by Anders Hejlsberg – father of C#
- Current version 2.6

Features

- 100% ECMA Script 3 or 5 support
- Static Typing
- Encapsulation using Revealing Module Pattern
- Support for constructors, properties, interfaces, enums
- Arrow Function support
- Can be combined with other JS Libs:
  - Angular
  - React
  - SharePoint Framework

Playground

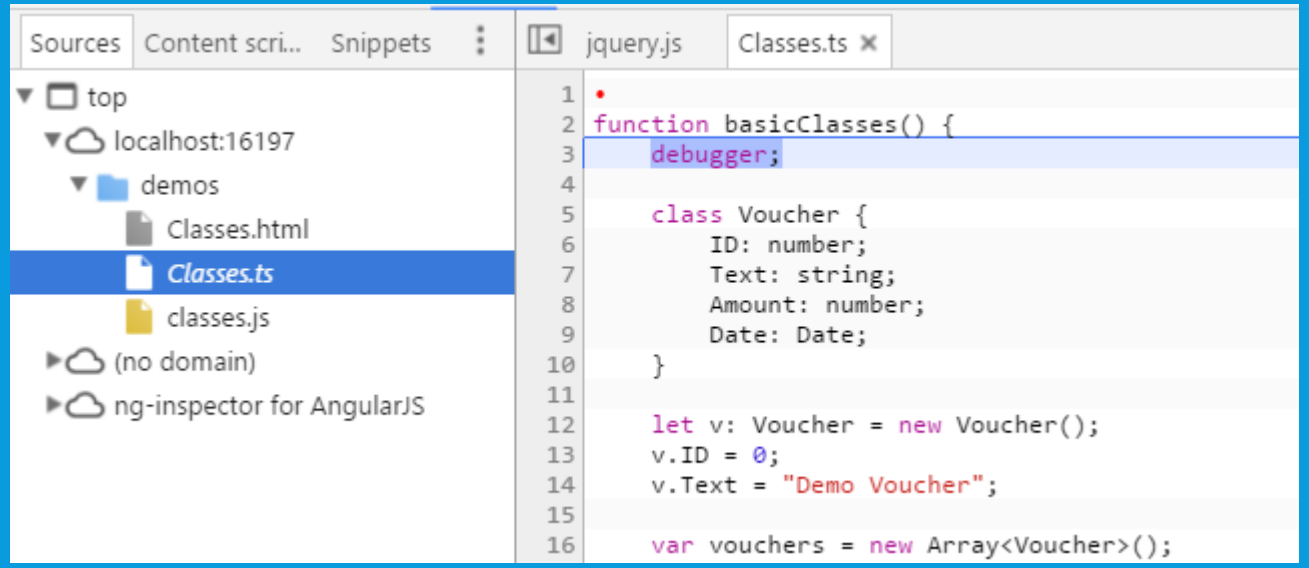
- Online editor at http://typescriptlang.org that helps understanding TS and see JS output

Keywords & Operators

Keyword	Description
class	Container for members such as properties and functions
constructor	Provides initialization functionality in a class
exports	Export a member from a module
extends	Extend a class or interface
implements	Implement an interface
imports	Import a module
interface	Defines a code contract that can be implemented by types
module	Container for classes and other code
public/private	Member visibility modifiers
...	Rest parameter syntax
=>	Arrow syntax used with definitions and functions
<typeName>	< > characters use to cast/convert between types
:	Separator between variable/parameter names and types

*.map files

- Map files are source map files that let tools map between the emitted JavaScript code and the TypeScript source files that created it
- Allows debugging *.ts files instead of the *.js files



Transpiling

- Transpiling is the process of converting TypeScript code to the requires ECMA Script version
- Can be automated to happen "on Save"

```
class Voucher {
    ID: number;
    Text: string;
    Amount: number;
    Date: Date;
}

let v: Voucher = new Voucher();
v.ID = 0;
v.Text = "Demo Voucher";
```

```
var Voucher = (function () {
    function Voucher() {
    }
    return Voucher;
}());

var v = new Voucher();
v.ID = 0;
v.Text = "Demo Voucher";
```

The Command Line Interface

- Typescript compilation (transpilation) is done using tsc.exe
- Converts TypeScript file to JavaScript file: tsc app.ts -> app.js
- Options documented @ https://www.typescriptlang.org/docs/handbook/compiler-options.html
- Configuration of tsc can be automated using tsconfig.json

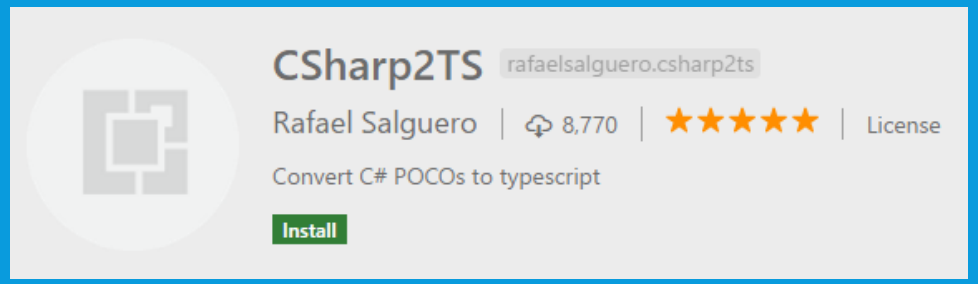
tsconfig.json

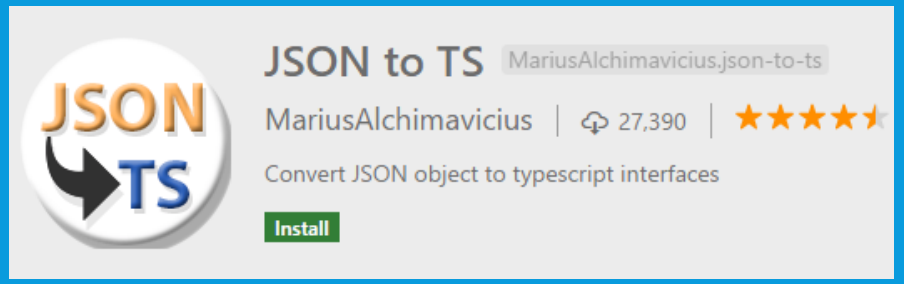
- Documented at https://www.typescriptlang.org/docs/handbook/tsconfig-json.html
- tsconfig.json file indicates and configures a TypeScript project.
- Files to be compiled can be configured using:
  - "files"
  - "include"
  - "exclude"

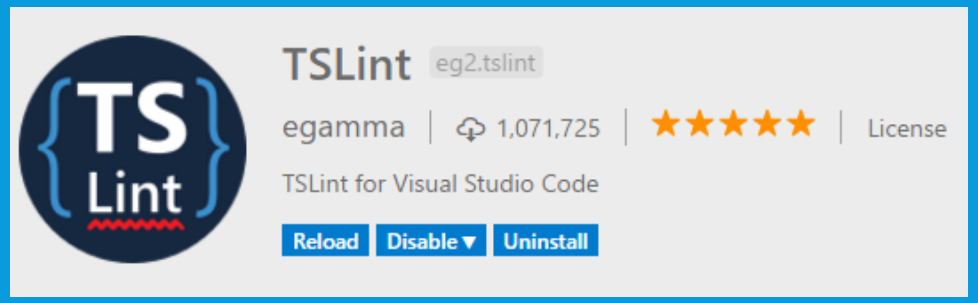
```
{
  "compileOnSave": true,
  "exclude": [
    "node_modules"
  ],
  "compilerOptions": {
    "target": "es5",
    "sourceMap": true,
    "rootDir": "./wwwroot/",
    "strict": false,
    "moduleResolution": "node",
    "experimentalDecorators": true
  }
}
```

VS Code Tools & Add-Ons

- CSharp2TS, JSON to TS-> Convert C# Classes (Data Models), JSON to Typescript
- TypeScript Hero … Auto Import







TS Lint

- Linting ensures Code Quality in teams & helps detect potential errors
- npm install -g tslint typescript.
- Configured using tslint.json - Documentation @ https://palantir.github.io/tslint/

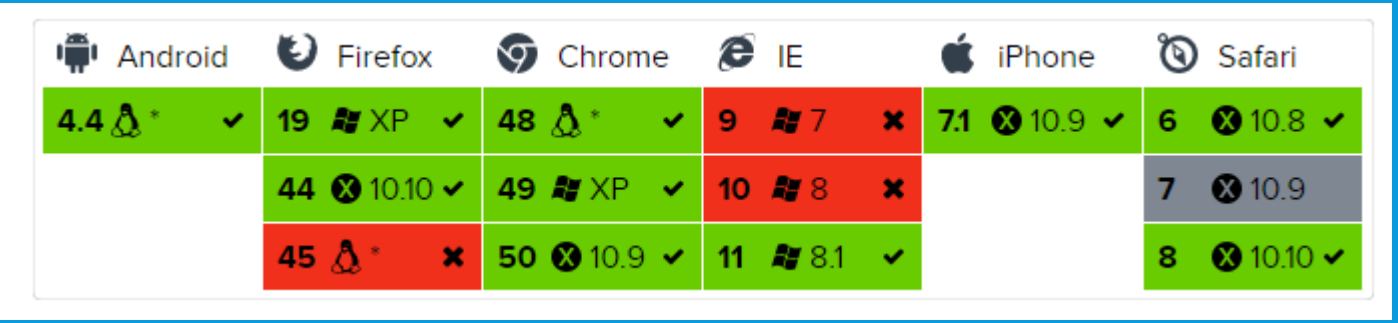
```
@Injectable()
export class Demo
    private items : DemoItem [] =[
        {url: "inline", title:'Inline Style'},
        {url: "stylebinding", title:'Style Binding'}
    ]
```

[tslint] expected nospace before colon in property-declaration (typedef-whitespace)

```
"typedef-whitespace": [
  true,
  {
    "call-signature": "nospace",
    "index-signature": "nospace",
    "parameter": "nospace",
    "property-declaration": "space",
    "variable-declaration": "space"
  }
],
```

ES 6 Shims

- Provides compatibility shims so that legacy JavaScript engines behave as closely as possible to ECMAScript 6
- A *shim* is a library that brings a new API to an older environment, using only the means of that environment
- Published @ https://www.npmjs.com/package/es6-shim



Polyfill

- A polyfill is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively.
- Thus, a polyfill is a shim for a browser API
- You typically check if a browser supports an API and load a polyfill if it doesn't

Types

Types and Variables

- string
- number
- Boolean
- any
- Date
- object, complex type
- void
- Null, undefinded

```
var age: number;
var weight: number = 83.12;
var dogWeight = 25.4;

var isCustomer: boolean = false;
var finished = false;

var dogName: string = "Giro";
var otherDogName = "Soi";
var x = 10;
```

Number, Strings & Booleans

- Boolean - The most basic datatype is the simple true/false value
- Number – Allows storage of all numeric types (descimal, hex, int, binary)
- String - Uses double quotes (") or single quotes (') to surround string data

```
var numbers: number[] = [];
numbers[0] = 1;
//numbers.push("two"); // compile-time error
```

var | let | const

- Variables can be declared using "var" or "let" – the difference is scoping
- "let" is scoped to the nearest enclosing block or global if outside any block
- const decrates constants – value cannot be changed

```
var index: number = 0;
var array = ["a", "b", "c"];
for (let index: number = 0; index < array.length; index++) {
    console.log("Inside for ..." + index);
    console.log("Inside for ..." + array[index]);
}
console.log(index); // 0
const pi = 3.14;
//pi = 2;
```

String Functions

- Template Literals using Backticks `...` and ${*VARIABLE*}
- String.prototype.repeat / String.prototype.contains
- String.prototype.startsWith / String.prototype.endsWith

```
//Template Literals
var productID = 100;
var category = "music";
var url = "http://server/" + category + "/" + productID;
var templateLiteral = `http://server/${category}/${productID}`;

//startsWith
var str = 'To be, or not to be, that is the question.';
console.log(str.startsWith('To be'));       // true
console.log(str.endsWith('question.'));     // true
```

Enums

- Enums allow us to define a set of named numeric constants.
- An enum can be defined using the enum keyword.

```
enum VoucherStatus {draft, complete, pending};

var n: VoucherStatus;
n = VoucherStatus.draft;
n = VoucherStatus.complete;
//n = VoucherStatus.unfinished; // compile-time error
//n = "on the way"; // compile-time error
```

Any

- Any allows you to gradually opt-in and opt-out of type-checking during compilation
- -> Avoid using Any
- Remember if you need to convert C# types to types in TypeScript use Typescript Syntax Past or one of the many online services

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

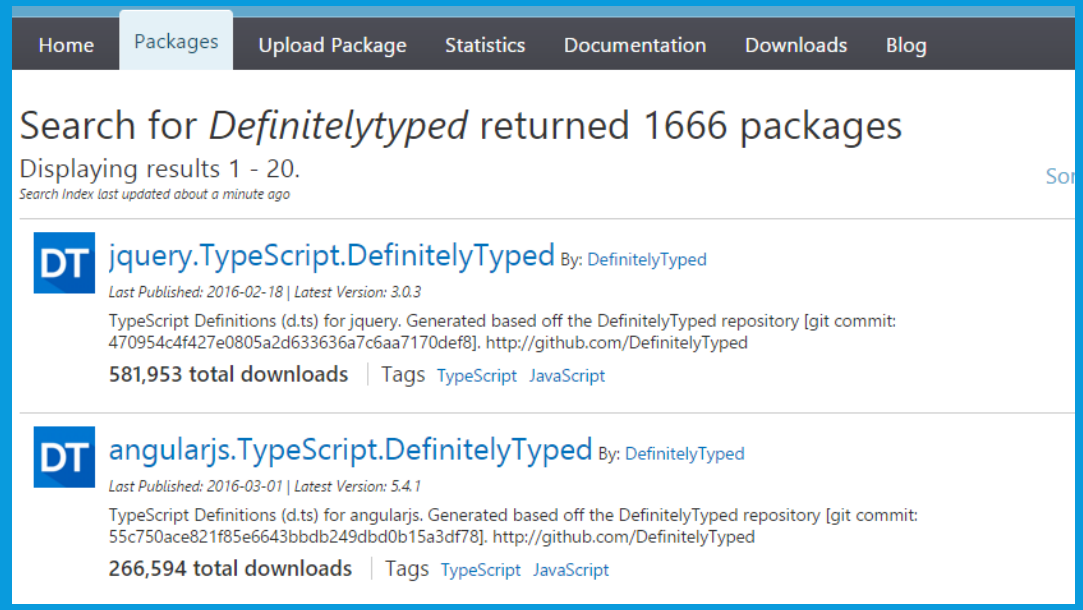
Void

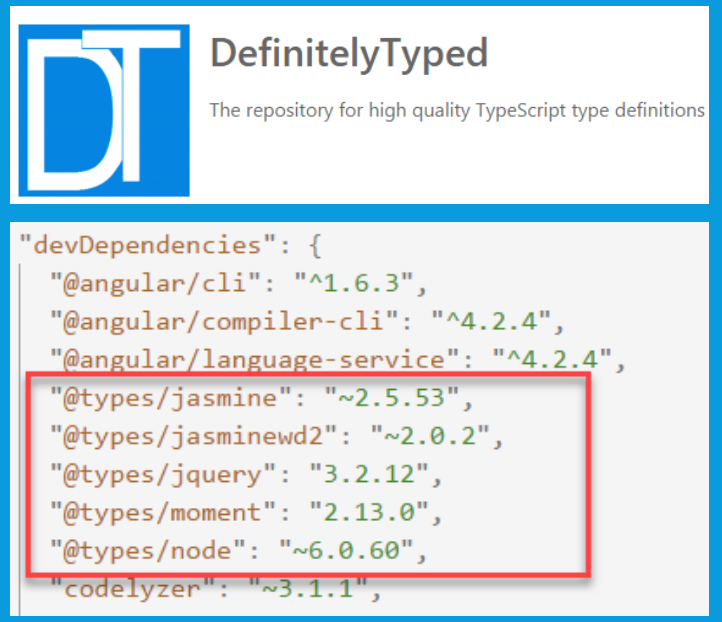
- Void is a little like the opposite of any – the absence of any type at all
- Usefull when used together with function
- No so usefull for variables -> can hold only null or undefined

```
function handleClick(): void {
    var g = "I don't return anything.";
    console.log(g);
}
```

Type Declarations

- Additional Types & IntelliSense for 3rd Party Libs using Type Definition Files like Definitely Typed published on http://definitelytyped.org/





Using Typings

- Add import statements as needed

```
import * as moment from 'moment';
import * as $ from 'jquery';
```

- Use your lib accordingly

```
//using moment
let dt = new Date();
console.log("Using time format: ", moment(dt).format('LTS'));

//using jQuery
let myArray = ["Angular", "React", "SPFx"];
console.log("myArray is an Array: ", $.isArray(myArray));
```

Arrays

Arrays

- Arrays can be typed

```
//declaration using type followed by []
var customers: string[] = ["Alex", "Giro", "Sonja", "Soi", "David"];
//declaration using generic array type
let nbrs: Array<number> = [3, 4, 5];
```

- ECMA Script 6 arry functionality is supported
  - for … of
  - destructuring
  - map
  - ….

```
let input = [1, 2];
let [first, second] = input;
```

For-of-loop

The for-of loop iterates over the values

- Of an array
- Of an iteratable object

```
var someArray = ["a", "b", "c"];
for (var item in someArray) {
    console.log(item); // 0,1,2 ... Returns the key ... the index
}

for (var item of someArray) {
    console.log(item); // a, b, c
}
```

Map

- The Map object is a simple key/value map.
- Any value (both objects and primitive values) may be used as either a key or a value.

```
var myMap = new Map();
var keyString = "a string",
    keyObj = {},
    keyFunc = function () { };
// setting the values
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, "value associated with keyObj");
myMap.set(keyFunc(), "value associated with keyFunc");
console.log("Map size: " + myMap.size); // 3
// getting the values
myMap.get(keyString);   // "value associated with 'a string'"
myMap.get("a string");  // "value associated with 'a string'" because keyString === 'a string'
myMap.get(keyObj);      // "value associated with keyObj"
```

Sets

- The Set object lets you store unique values of any type, whether primitive values or object references.
- Can iterate its elements in insertion order. A value in the Set may only occur once; it is unique in the Set's collection.

```
var mySet = new Set();
mySet.add(1);
mySet.add("some text");
var o = { a: 1, b: 2 };
mySet.add(o);

mySet.has(1); // true
mySet.has(3); // false, 3 has not been added to the set
mySet.has(Math.sqrt(25));  // true
mySet.has("Some Text".toLowerCase()); // true
mySet.has(o); // true
mySet.size; // 4
mySet.delete(5); // removes 5 from the set
```

Destructuring

- Makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
var rect = { x: 0, y: 10, width: 15, height: 20 };

// Destructuring assignment
var { x, y, width, height } = rect;
console.log(x, y, width, height); // 0,10,15,20
```

REST Parameter

- Represented using ...items
- Allows calling a function with a variable numer of arguments without using the arguments object

```
store.add('fruit', 'apple');
store.add('dairy', 'milk', 'cheese', 'yoghurt');
store.add('pastries', 'donuts', 'croissants');

store.add = function(category, ...items) {
    items.forEach(function (item) {
        store.aisle[category].push(item);
    });
};
```

Spread Operator

- The spread operator allows an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) are expected

```
var a, b, c, d, e;
a = [1, 2, 3];
b = "dog";
c = [42, "cat"];

// Using the concat method.
d = a.concat(b, c);
// Using the spread operator.
e = [...a, b, ...c];
console.log(d);
console.log(e);

// Output:
// 1, 2, 3, "dog", 42, "cat"
// 1, 2, 3, "dog", 42, "cat"
```

Objects

Object Literals

- An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces {}
- Possible to add funtions to Object Literals

```
let person: any = {Id: 1, Name: 'Alexander'}
person.walk = () => console.log(`I am ${person.Name} and I'm walking`);
```

Property / Method Shorthand

```
//Property value shorthand
function getCarES5(make, model, value) {
 return {
  make: make,
  model: model,
  value: value
 };
}

// with property value shorthand
// syntax, you can omit the property
// value if key matches variable
// name
function getCar(make, model, value) {
 return {
  make,
  model,
  value
 };
}
```

```
//Method definition shorthand
function getBusES5(make, model, value) {
 return {
  depreciate: function() {
   this.value -= 2500;
  }
 };
}

// Method definition shorthand syntax
// omits `function` keyword & colon
function getBus(make, model, value) {
 return {
  depreciate() {
   this.value -= 2500;
  }
 };
}
```

Value / Reference Types

- Javascript is always pass by value, but when a variable refers to an object (including arrays), the "value" is a reference to the object.
- Changing the value of a variable never changes the underlying primitive or object, it just points the variable to a new primitive or object.
- However, changing a property of an object referenced by a variable does change the underlying object.

Immutability

- In object-oriented programming, an immutable object is an object whose state cannot be modified after it is created.
- In JS string and numbers are immutable by design for other objects use Immutable.js
- The concept of Immutability is often used in larger applications when State Management is centralized using libraries like Redux (… which can be used to manage State in Angular Apps)
- Immutability eliminates the risk of having unwanted side effects by always creating „new" copies of objects when changing the state

object.assign()

- Is used to copy the values of all enumerable own properties from one or more source objects to a target object.
- Will return the target object
- Used when working with architectural patterns like Redux & immutablity

```
var obj = { name: 'alex' };
var copy = Object.assign({}, obj, {birth: moment("19700402", "YYYYMMDD").format("MMM Do YY")});
console.log(copy);
```

Cloning Objects using Spread Operator

- Spread Operator can be used to
  - create „new" copies of objects or
  - combine a set of objects
- Needs no polyfill for older browsers (… compared to object.assign())

```
var simplePerson = { name: 'alex' };
var dataPerson = {birth: moment("19700402", "YYYYMMDD").format("MMM Do YY"), job: 'dev dude'}

var person = {...simplePerson, ...dataPerson};
console.log(person);
```

Functions

Functions

- Typescript knows named and anonymous functions
- Parameteres can be typed
- Optional, default parameters supported
- Lambda Expressions, Rest Parameters supported

```
function multiply(a, b = 1) {
    return a*b;
}

multiply(5); // 5
```

```
var rectangleFunction = function (width: number, height: number) {
    return width * height;
}

//Implemented as Lambda or "Arrow" Function
var rectangleFunctionArrow = (width: number, height: number) => height * width;
var result: number = rectangleFunctionArrow(10, 22);
```

Arrow Functions

- Known als Lambda Functions in C#

```
numbers.sort(function(a, b){
    return b - a;
});
```

Becomes

```
numbers.sort((a, b) => b - a);
```

```
var fullnames = people.filter(p => p.age >= 18).map(p => p.fullname);
```

Function Overloading

- TypeScript allows you to define overloaded functions
- Only one impelmentation
- Requires type checking during the implementation because of underlying JS

```
addCustomer(custId: number){};
addCustomer(company: string);
addCustomer(value: any) {
    if (value && typeof value == "number") {
        alert("First overload - " + value);
    }
    if (value && typeof value == "string") {
        alert("Second overload - " + value);
    }
}
```

Generator Functions

- Functions that can be called multible times until they are executed to thie final stage
- Each "yield"-statement defines one stop in execution
- Many times used together with for ... of to generate data

```
function* getColors() {
    //Code to be executed in between
    yield "green";
    yield "red";
    yield "blue";
}

const colorGenerator = getColors();

debugger;
console.log(colorGenerator.next());
console.log(colorGenerator.next());
console.log(colorGenerator.next());
```

Classes, Interfaces

Classes

Implemented as modules to avoid poluting global namespace with the following members

Support:

- Fields – referenced using "this." – e. g. this.greeting
- Properties
- Constructor
- Functions
- ….

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

var greeter = new Greeter("world");
```

Constructor

- Called when creating an instance of a class
- Canbe overloaded – but only with one implementation
- Can define public and private properties
- Can define default values and nullable paremeters

```
class Person {
    name: string;
    alive: boolean;
    constructor(Name: string, Alive: boolean) {
        this.name = Name;
        this.alive = Alive;
    }
}
```

get / set

```
let passcode = "secret passcode";

class Citzien {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode == "secret passcode") {
            this._fullName = newName;
            console.log("name changed to " + newName);
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}
```

Class Inheritance

- Class inheritance is archieved using the "extends" keyword
- Protected / Private / ReadOnly Properties are supported - Abstract classes are supported
- Properties of the base calls are accessed using "super"

```
class Sighthound extends Dog {
    constructor(name: string) { super(name); }
    public speed: string = "with up to 110 km/h";
    move(meters = 500) {
        console.log("Running ..." + meters + "m. " + this.speed);
        super.move(meters);
    }
}
```

Static Members (Properties)

- Classes in TypeScript can either contain
  - static members or
  - instance members.

```
class Grid {
    static origin = { x: 0, y: 0 };
    calculateDistanceFromOrigin(point: { x: number; y: number; }) {
        var xDist = (point.x - Grid.origin.x);
        var yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }
    constructor(public scale: number) { }
}
```

Interfaces

- In TypeScript, interfaces fill the role of defining contracts within your code as well as contracts with code outside of your project
- Support optional properties
- Can also be used to describe functions

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}

var mySearch: SearchFunc;
mySearch = function (source: string, subString: string) {...}
```

Nullablility

- Interfaces support nullable properties

```
interface IManager {
    name: string;
    salary?: number;
}

class DeliveryManager implements IManager {
    name: string;
}
```

Interfaces and Objects

- Interfaces are many times used to hold value objects
  - Data received from an WebApi or Angular service
  - Client should only know structure of data

```
interface ILongLat { Long: number, Lat: number };
var position: ILongLat = { Long: 17.123123, Lat: 12.123123 };
console.log("We are at position Long: " + position.Long + " Lat: " + position.Lat);
```

Generics, Modules, Decorators

Generic Functions

- Generics allow creating reusable components that can return any given type
- The type in the generic is passed using <T>

```
function concat<T>(arg: Array<T>): string {
    let result = "";
    for (var m of arg) {
        result += m.toString() + ", ";
    }
    return result;
}

let stringArr: Array<string> = ["Alex", "Giro", "Soi the Whippet"];
console.log(concat<string>(stringArr));

let nbrArr: Array<number> = [100, 201, 322];
console.log(concat<number>(nbrArr));
```

Generic Interfaces

- Generic Interfaces define Interfaces for a give type specified using T

```
interface IInventory<T> {

    getNewestItem: () => T;
    addItem: (newItem: T) => void;
    getAllItems: () => Array<T>;
}

let voucherInventory: IInventory<Vouchers.IVoucher>;
```

Generic Classes

- Used to implement utility classes for a given type
- Can implement generic interfaces

```
class Catalog<T> implements IInventory<T> {

    private items = new Array<T>();

    addItem(newItem: T) { this.items.push(newItem);}

    getNewestItem(): T { return this.items[this.items.length-1]; }

    getAllItems(): T[] { return this.items; }
}
```

Generic Constraints

- Describe types that may be passed as a generic parameter
- The "extends"-keyword applies the constraint

```
interface ICatalogItem {
    catalogNumber: number;
}

interface IInventory<T> {
    getAllItems: () => Array<T>;
}

class Catalog<T extends ICatalogItem> implements IInventory<T> {
    private items = new Array<T>();
    getAllItems(): T[] { return this.items; }
}
```

Modules

- Organize other elements like classes and interfaces for better maintainability
- Are executed in their own scope – not the global scope
- Elements that should be visible outside must explicitly be exported / imported

```
export namespace MathFunctions {
    export function square(nbr: number): number {
        return Math.pow(nbr, 2);
    }
}
```

mathFunctions.ts

```
import mathFunctions = require("./mathFunctions");
let sq = mathFunctions.MathFunctions.square(10);
```

otherFile.ts

Namespaces

- Namespaces are declared using "namespace" keyword
- TypeScript allows multi-file namespaces
- Aliases simplify working with namespaces

```
export namespace MathFunctions {
    export function square(nbr: number) : number {
        return Math.pow(nbr, 2);
    }
}

import  mf = MathFunctions;
let sq = mf.square(10);
```

Decorators

- Decorators add descriptive information to classes
  - Annotations
  - Metadata
  - Used by frameworks like
    - Angular
    - React

```typescript
import {Component} from '@angular/core';

@Component({
  selector: 'app',
  templateUrl: './app.component.html',
})
export class AppComponent {
}
```

tsconfig.json

```json
{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "baseUrl": "src",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2016",
      "dom"
    ]
  }
}
```

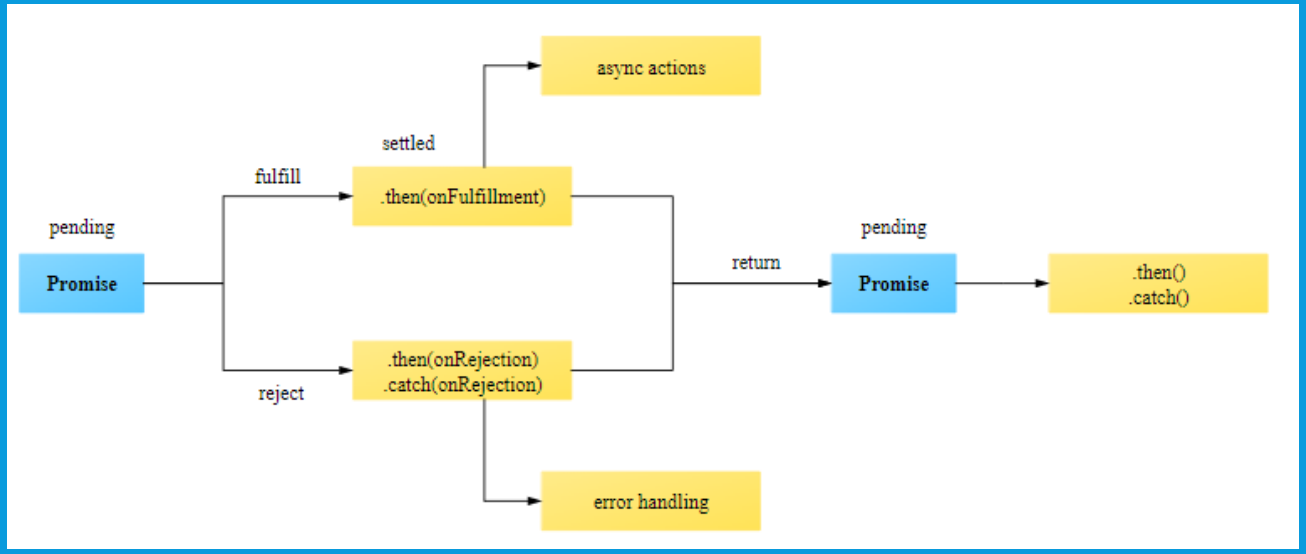
Consuming Services

Consuming Services Overview

- Service calls are typically async
- ES 6 fetch @ moment only supported by Chrome -> $.ajax
- 3 implementation paths
  - $.ajax -> success
  - jQuery Deferred
  - await

ES 6 Promise

- Pattern to deal with async
- 3 States: pending (in progress), fulfilled (success), rejected (error)
- Use .then() & .catch() for further processing and error handling



Using ES6 Promises

- ES6 defines Promise
- If transpiling to ES5 reference ES6-shim (https://github.com/paulmillr/es6-shim)

```
function doAsyncTask(succeed: boolean): Promise<string> {
    return new Promise<string>((resolve, reject) => {
        setTimeout(() => {
            console.log("Async Task Complete");
            if (succeed) {resolve("Outcome: Promise resolved");}
            else {reject("Outcome: Promise rejected");}
        }, 1000);
    });
}

doAsyncTask(true).then((msg) => {
    console.log(msg);
});
```

Using fetch & async / await

- ES 6 offeres Fetch API for interacting with HTTP pipeline
- fetch-polyfill available for ES5 (https://www.npmjs.com/package/fetch-polyfill)
- await – task pattern from C# implemented in TypeScript -> clearer coding

```
async function getAllVouchers() {
    let response = await fetch("./demos/vouchers.json");
    let voucher = await response.json();
    console.log("Data received");
    console.log(voucher);
}

getAllVouchers();
```

Links & Ressources

- Typescript Website - http://www.typescriptlang.org/docs
- Samples- https://github.com/Microsoft/TypeScript-Handbook/tree/master/pages
- Book - https://www.gitbook.com/book/basarat/typescript/details
- Cheatsheet - https://www.sitepen.com/blog/2013/12/31/typescript-cheat-sheet/

Demo

Using TypeScript

- VouchersTypeScript -> Demos -> demo.html