TypeScript

Agenda

TypeScript Overview

Config tsc & VSCode

Types & Arrays, Functions

ECMA Script 6+ Essentials

Objects, Classes, Interfaces

Generics, Modules, Decorators

Consuming Services

TypeScript Overview

What is TypeScript

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript

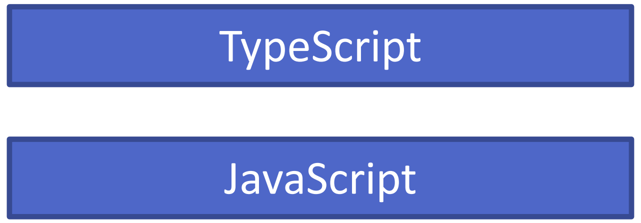
Allows use of classes and other features in browsers that do not support ECMA Script

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 4.0



TypeScript vs JavaScript

JavaScript Standard is defined as ECMA Script

ES 6 introduced modern JavaScript

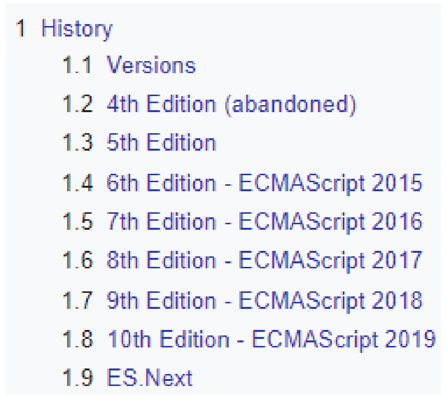
- Classes, Decorators
- Array Helpers
- Arrow Functions
- Fetch API

Typescript adds on Top

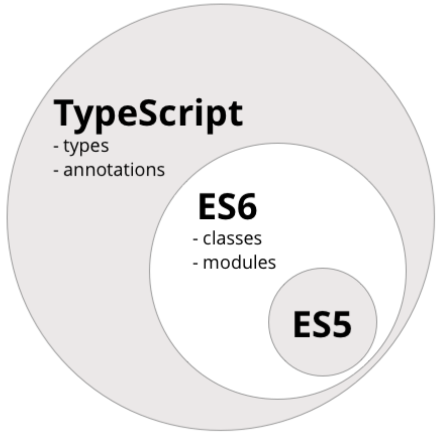
- Types, Enums, Interfaces ....

Angular 8+ supports Differential loading

- ES5 vs ES6



1 History
1.1 Versions
1.2 4th Edition (abandoned)
1.3 5th Edition
1.4 6th Edition - ECMAScript 2015
1.5 7th Edition - ECMAScript 2016
1.6 8th Edition - ECMAScript 2017
1.7 9th Edition - ECMAScript 2018
1.8 10th Edition - ECMAScript 2019
1.9 ES.Next



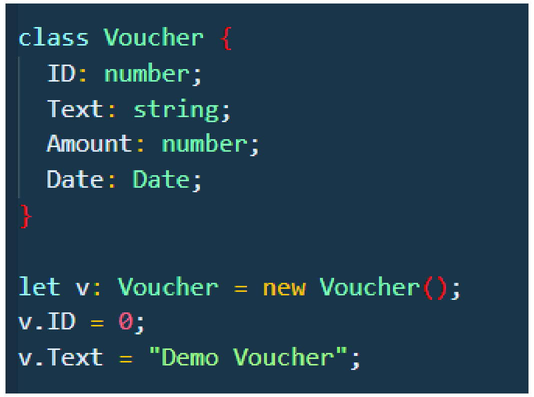
Transpiling

Transpiling is the process of converting TypeScript code to the requires ECMA Script version

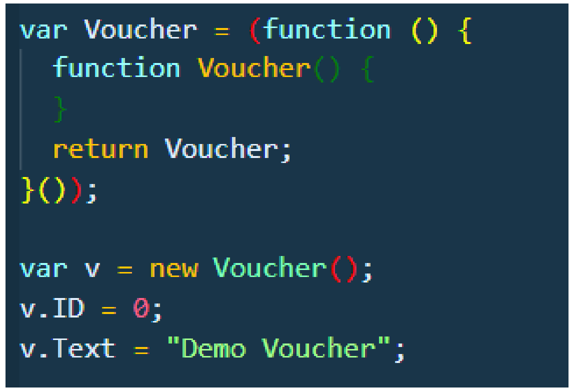
Done (in Background) using tsc.exe ->tsc app.ts -> app.js

Configuration of tsc can be automated using tsconfig.json

Options @ https://www.typescriptlang.org/docs/handbook/compiler-options.html



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

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

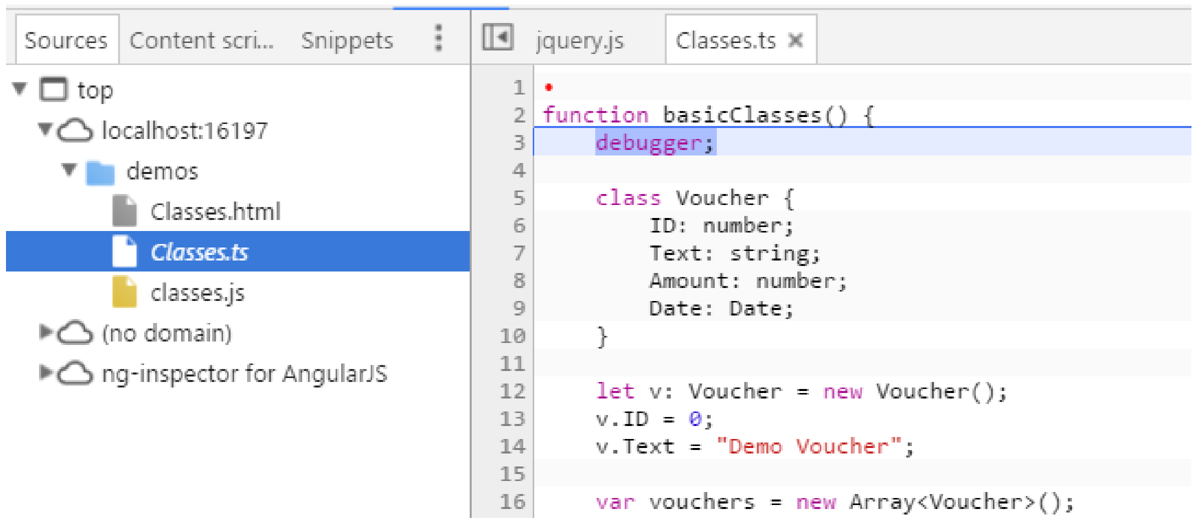


```
var Voucher = (function () {
  function Voucher() {
  }
  return Voucher;
}());

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

Source Maps

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



```

function basicClasses() {
    debugger;

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

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

    var vouchers = new Array<Voucher>();
```

tsconfig.json

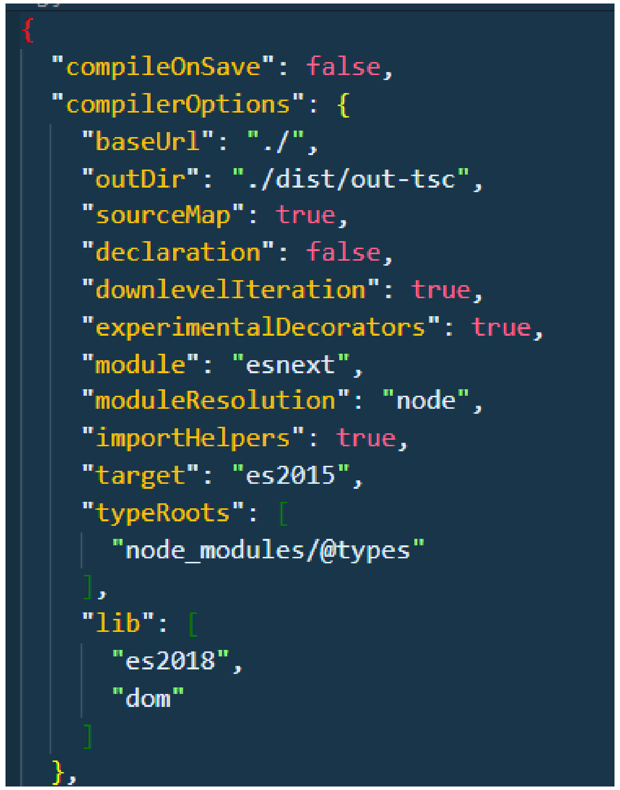
Sets the compiler options used by tsc.exe

Indicates and configures a TypeScript project

- lib sets ECMA Script Version

Files to be compiled can be configured using:

- "files" | "include" | "exclude"



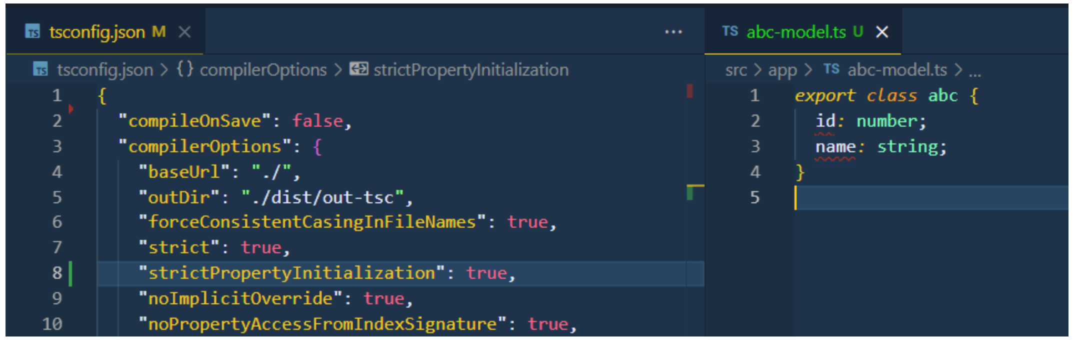
```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "module": "esnext",
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "es2015",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "dom"
    ]
  },
```

Strict Mode

Strict mode improves maintainability and helps you catch bugs ahead of time

Turns on strict Angular compiler flags

- strictTemplates,
- strictInjectionParameters
- strictInputAccessModifiers


tsconfig.json M
abc-model.ts U
tsconfig.json > {} compilerOptions > strictPropertyInitialization
src > app > abc-model.ts > ...

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "strictPropertyInitialization": true,
    "noImplicitOverride": true,
    "noPropertyAccessFromIndexSignature": true,
```

```
export class abc {
  id: number;
  name: string;
}
```

StrictPropertyInitialization is turned on / off seperately

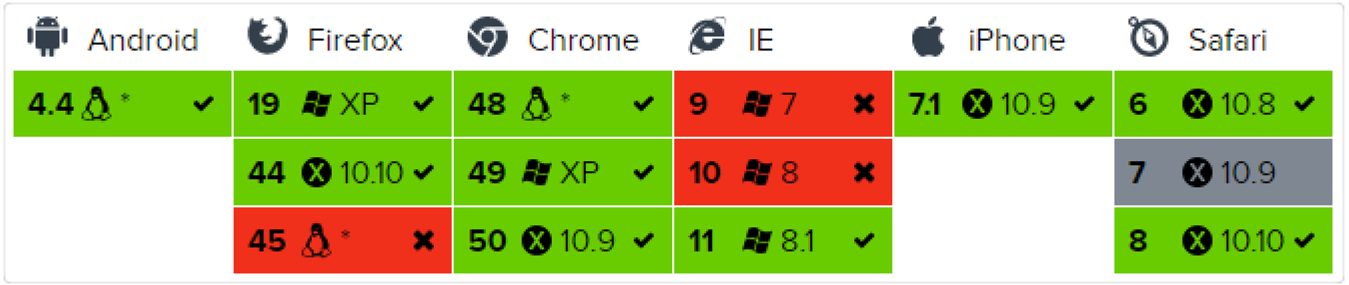
More info @ *https://angular.io/config/tsconfig*

Shims & Polyfills

Provides compatibility so that legacy JavaScript Engines (Old Browsers) behave as closely as possible to ECMAScript 6

A *Shim* or Polyfill is a library that enables a new API on an older environment, using only means of that environment

Published @ https://www.npmjs.com/package/es6-shim



Android	Firefox	Chrome	IE	iPhone	Safari
4.4 * ✓	19 XP ✓	48 * ✓	9 7 ✗	7.1 10.9 ✓	6 10.8 ✓
	44 10.10 ✓	49 XP ✓	10 8 ✗		7 10.9
	45 * ✗	50 10.9 ✓	11 8.1 ✓		8 10.10 ✓

ES Lint

Replaces TS Lint - Not installed by default in Angular Projects

Must be installed manually using:

- ng add @angular-eslint/schematics

Existing TS Lint can be converted using:

- ng g @angular-eslint/schematics:convert-tslint-to-eslint

Types

Types and Variables

string

number

boolean

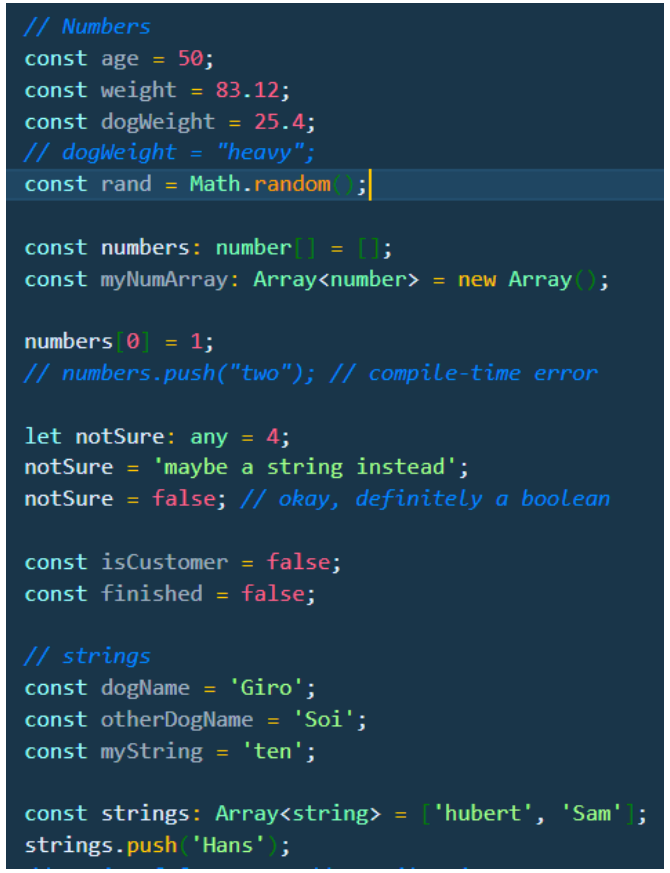
any

Date

object, complex type

void

Null, undefinded



```
// Numbers
const age = 50;
const weight = 83.12;
const dogWeight = 25.4;
// dogWeight = "heavy";
const rand = Math.random();

const numbers: number[] = [];
const myNumArray: Array<number> = new Array();

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

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

const isCustomer = false;
const finished = false;

// strings
const dogName = 'Giro';
const otherDogName = 'Soi';
const myString = 'ten';

const strings: Array<string> = ['hubert', 'Sam'];
strings.push('Hans');
```

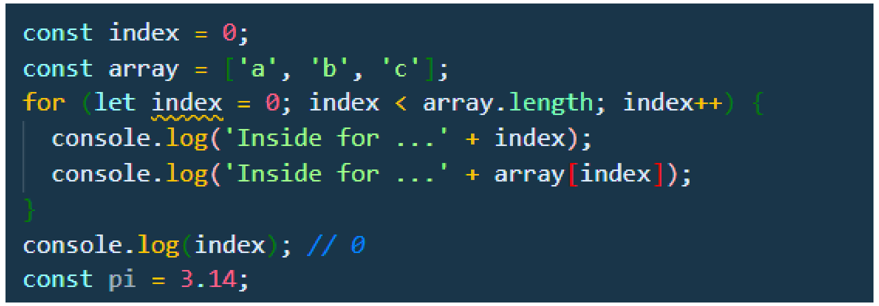
var | let | const

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
- let is the prefered approach -> no-var lint rule

const declares constants – value cannot be changed

- if re-assignment is not intended use const -> prefer-const lint rule
- Decoaring a var twice like in the example is not recommended
  - breaks no-shadowed-variable lint rule



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

Any

Any allows you to gradually opt-in and opt-out of type-checking during compilation

- Avoid using 'any'
- Angular stric mode settings turn on no-any tslint rule to prevent declarations of type 'any'

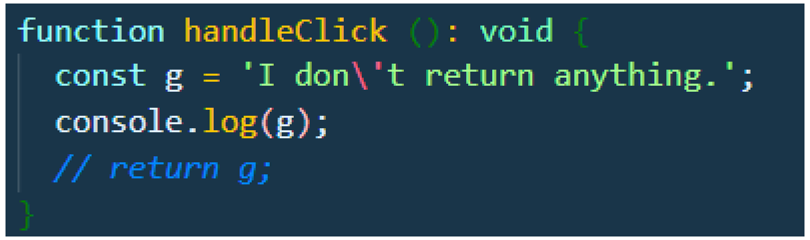
If you need to convert C# types to TypeScript:

- Use Typescript Syntax Paste or one of the many online services

void

void is a little like the opposite of any – the absence of any type at all

- Usefull when used together with function
- Not so usefull for variables -> can hold only null or undefined
- Angular strict mode requires all methods to set a return type even it is void



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

Number, Strings & Booleans

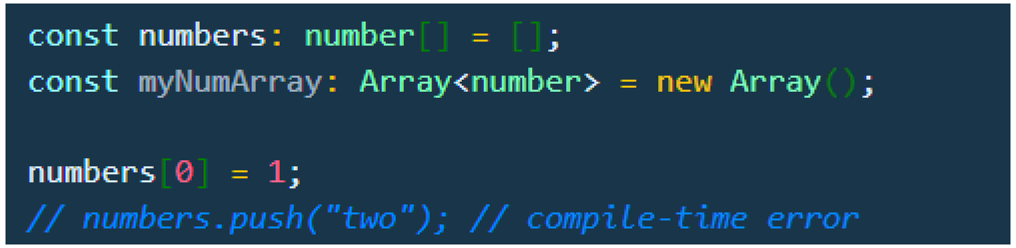
boolean - The most basic datatype is the simple true/false value

number – Allows storage of all numeric types (descimal, hex, int, binary)

string - Uses double quotes (") or single quotes (') to surround string data

Do not use String, Number or Boolean – these are objects

- Always use string, number, boolean



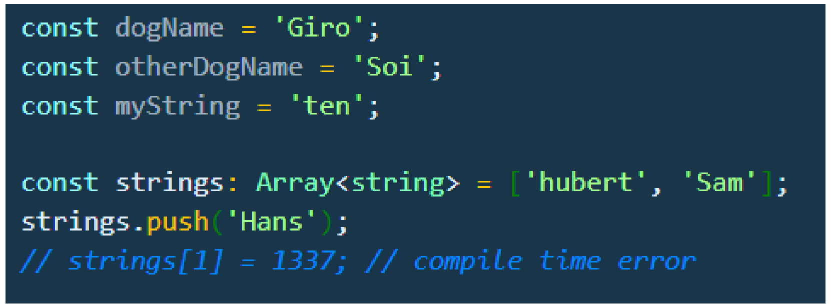
```
const numbers: number[] = [];
const myNumArray: Array<number> = new Array();

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

String Functions

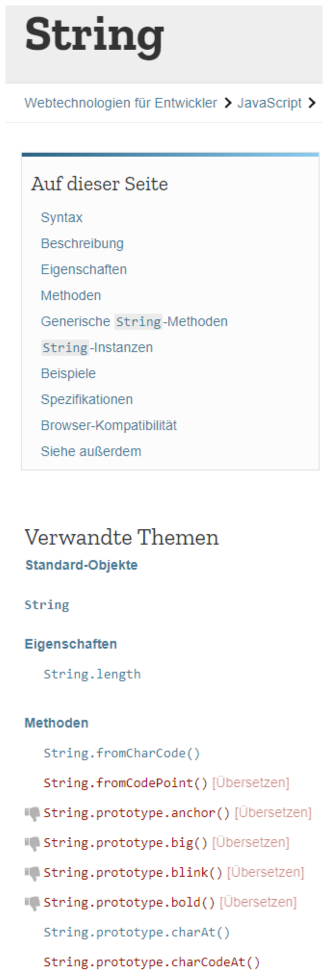
Template Literals using Backticks `...` and ${*VARIABLE*}

String.repeat / String.contains

String.startsWith / String.endsWith



```
const dogName = 'Giro';
const otherDogName = 'Soi';
const myString = 'ten';

const strings: Array<string> = ['hubert', 'Sam'];
strings.push('Hans');
// strings[1] = 1337; // compile time error
```



String

Webtechnologien für Entwickler > JavaScript >

Auf dieser Seite

- Syntax
- Beschreibung
- Eigenschaften
- Methoden
- Generische `String`-Methoden
- `String`-Instanzen
- Beispiele
- Spezifikationen
- Browser-Kompatibilität
- Siehe außerdem

Verwandte Themen

Standard-Objekte

`String`

Eigenschaften

`String.length`

Methoden

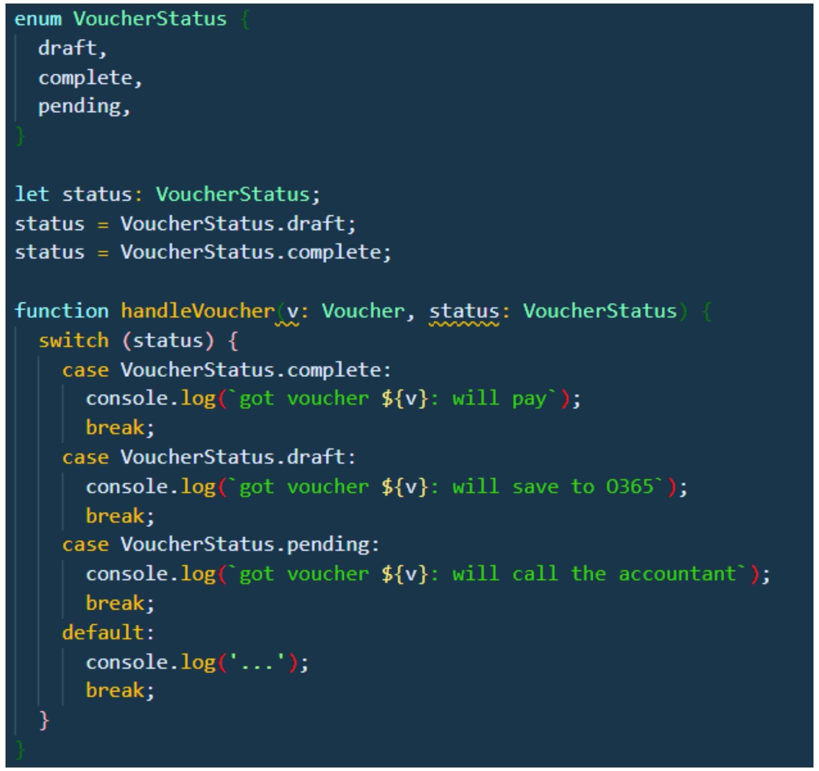
- `String.fromCharCode()`
- `String.fromCodePoint()` [Übersetzen]
- `String.prototype.anchor()` [Übersetzen]
- `String.prototype.big()` [Übersetzen]
- `String.prototype.blink()` [Übersetzen]
- `String.prototype.bold()` [Übersetzen]
- `String.prototype.charAt()`
- `String.prototype.charCodeAt()`

Enums

Enums allow us to define a set of named constants of type

- string
- number

Defined using the enum keyword.



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

let status: VoucherStatus;
status = VoucherStatus.draft;
status = VoucherStatus.complete;

function handleVoucher(v: Voucher, status: VoucherStatus) {
  switch (status) {
    case VoucherStatus.complete:
      console.log(`got voucher ${v}: will pay`);
      break;
    case VoucherStatus.draft:
      console.log(`got voucher ${v}: will save to O365`);
      break;
    case VoucherStatus.pending:
      console.log(`got voucher ${v}: will call the accountant`);
      break;
    default:
      console.log('...');
      break;
  }
}
```

Ambient Declarations

A way of telling the TypeScript compiler that the actual source code of an object / variable exists elsewhere or is present in the Global Namespace

Used when consuming objects from third party libs

Syntax:

- declare module NAME
- declare var NAME

Type Defintions – "Typings"

Used to provide Types & IntelliSense for 3rd Party JavaScript libs

- Most 'modern' ES-Modules include Typings

Implemented using *.d.ts-Files

Initially published on http://definitelytyped.org/

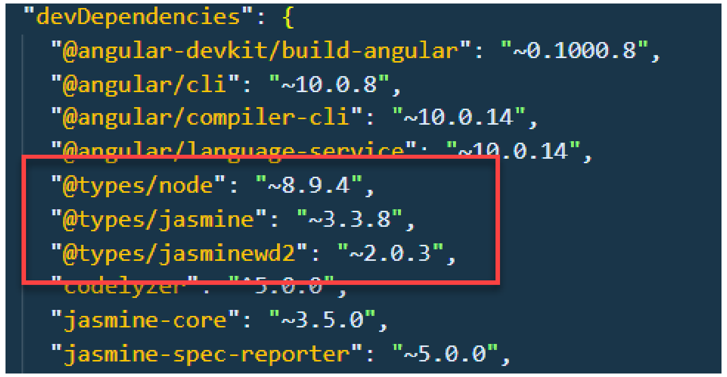
Also available from Typings, NPM

Today:

- Referenced in package.json as devDependency

DefinitelyTyped

The repository for high quality TypeScript type definitions

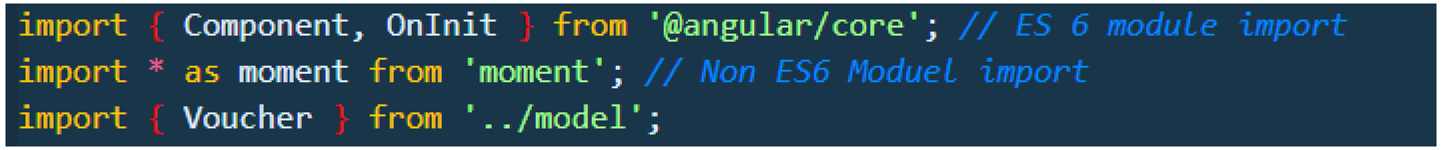


```
"devDependencies": {
  "@angular-devkit/build-angular": "~0.1000.8",
  "@angular/cli": "~10.0.8",
  "@angular/compiler-cli": "~10.0.14",
  "@angular/language-service": "~10.0.14",
  "@types/node": "~8.9.4",
  "@types/jasmine": "~3.3.8",
  "@types/jasminewd2": "~2.0.3",
  "codelyzer": "^5.0.0",
  "jasmine-core": "~3.5.0",
  "jasmine-spec-reporter": "~5.0.0",
```

Working with 3rd Party Libs

Use the following steps when working with 3rd Party Libs

- npm install package-name --save
- npm install @types/thePackageName --save
- import { function } from 'packageName’;



```
import { Component, OnInit } from '@angular/core'; // ES 6 module import
import * as moment from 'moment'; // Non ES6 Moduel import
import { Voucher } from '../model';
```

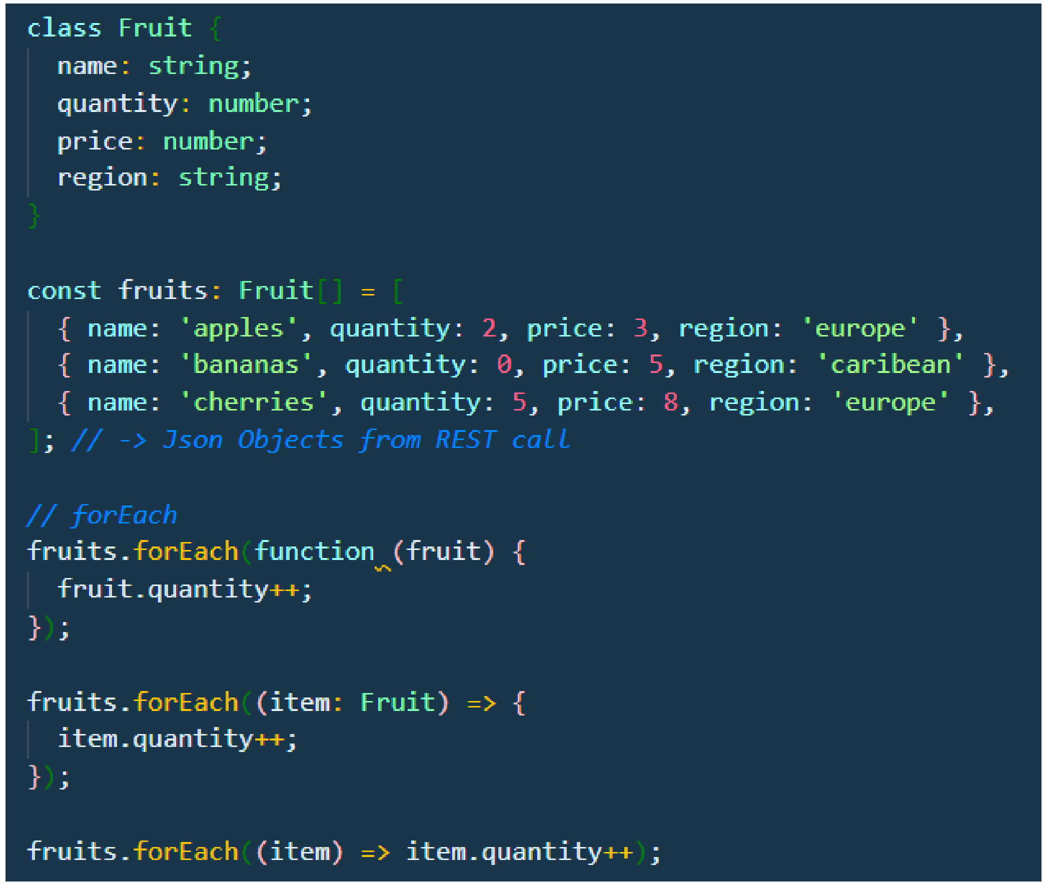
Arrays

Arrays

Arrays in Typescript can be typed

ECMA Script 6 array functionality is supported

- for ... of
- find / filter
- destructuring
- map
- ....

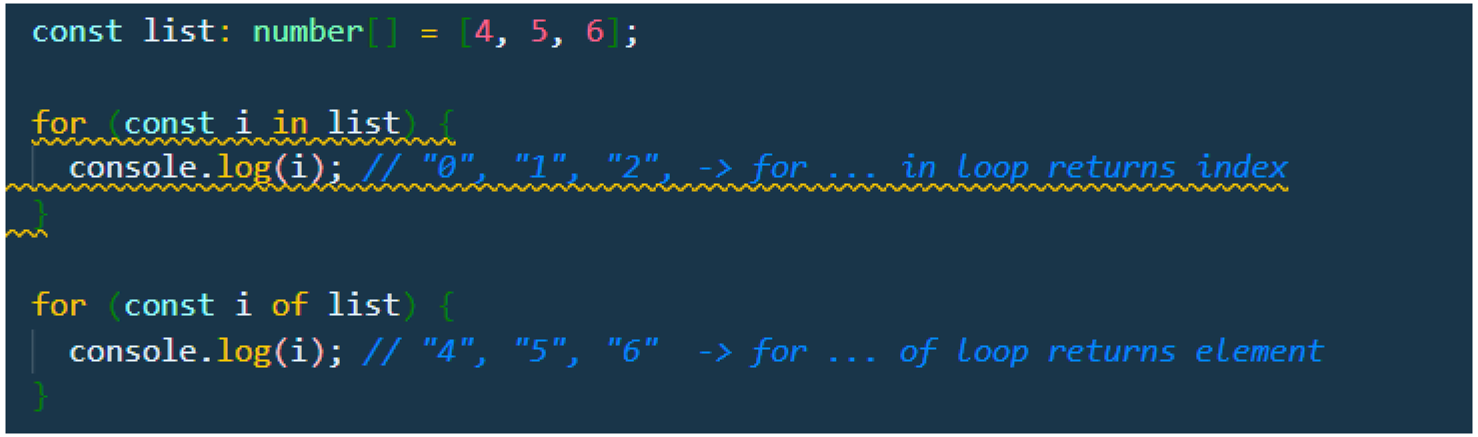


```
class Fruit {
  name: string;
  quantity: number;
  price: number;
  region: string;
}

const fruits: Fruit[] = [
  { name: 'apples', quantity: 2, price: 3, region: 'europe' },
  { name: 'bananas', quantity: 0, price: 5, region: 'caribean' },
  { name: 'cherries', quantity: 5, price: 8, region: 'europe' },
]; // -> Json Objects from REST call

// forEach
fruits.forEach(function (fruit) {
  fruit.quantity++;
});

fruits.forEach((item: Fruit) => {
  item.quantity++;
});

fruits.forEach((item) => item.quantity++);
```

For-of-loop

The for-of loop iterates over the values

- Of an array
- Of an iteratable object
- Avoid for-in loop
  - ts-lint rule: forin



```
const list: number[] = [4, 5, 6];

for (const i in list) {
  console.log(i); // "0", "1", "2", -> for ... in loop returns index
}

for (const i of list) {
  console.log(i); // "4", "5", "6"  -> for ... of loop returns element
}
```

Map

The Map object is a simple key/value map.

- Like a C# dictionary
- Better performance than array when accessing items

Any value (both objects and primitive values) may be used as either a key or a value.

Sets

The Set object lets you store unique values of any type, whether primitive values or object references.

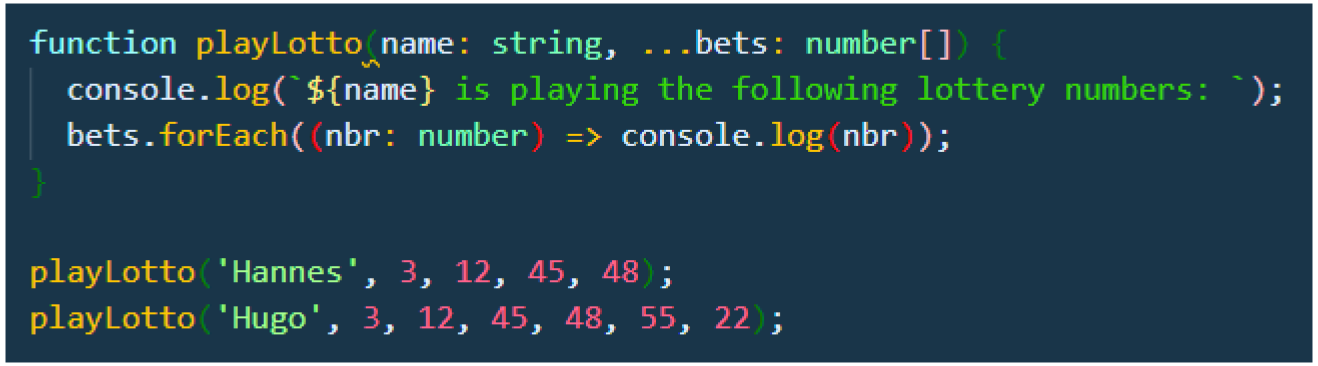
Can iterate its elements in insertion order.

A value in the Set may only occur once; it is unique in the Set's collection.

REST Parameter

Represented using ...items

Allows calling a function with a variable number of arguments without using the arguments object

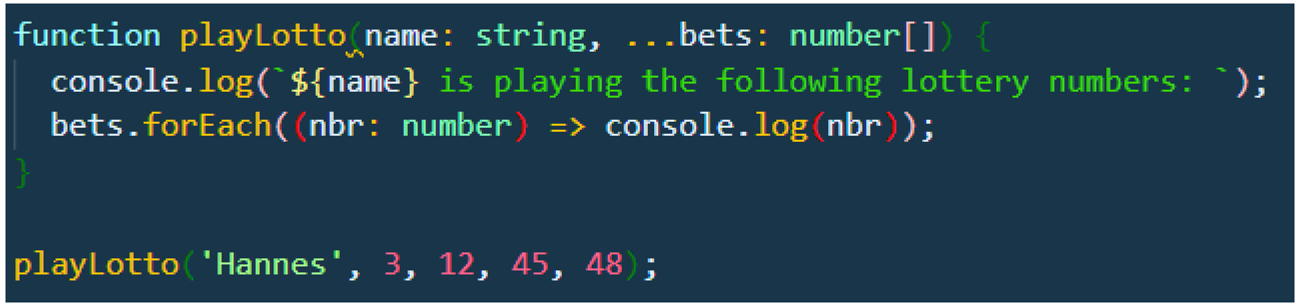


```
function playLotto(name: string, ...bets: number[]) {
  console.log(`${name} is playing the following lottery numbers: `);
  bets.forEach((nbr: number) => console.log(nbr));
}

playLotto('Hannes', 3, 12, 45, 48);
playLotto('Hugo', 3, 12, 45, 48, 55, 22);
```

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

Expands a copy of the object on this level



```
function playLotto(name: string, ...bets: number[]) {
  console.log(`${name} is playing the following lottery numbers: `);
  bets.forEach((nbr: number) => console.log(nbr));
}

playLotto('Hannes', 3, 12, 45, 48);
```

Functions

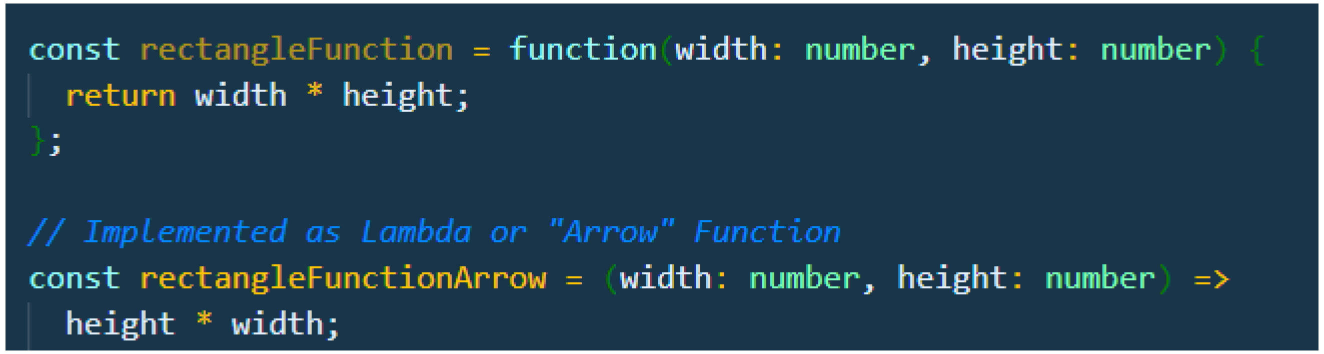
Functions

Named and Anonymous functions can return Typed values

Parameteres can be typed

Optional, default parameters supported

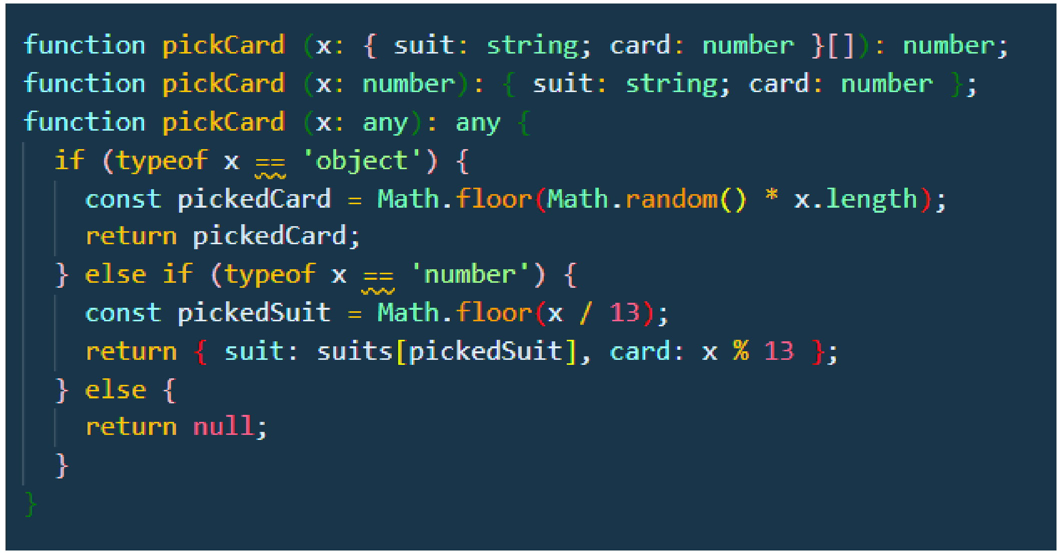
Lambda Expressions, Rest Parameters supported



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

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

Function Overloading

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



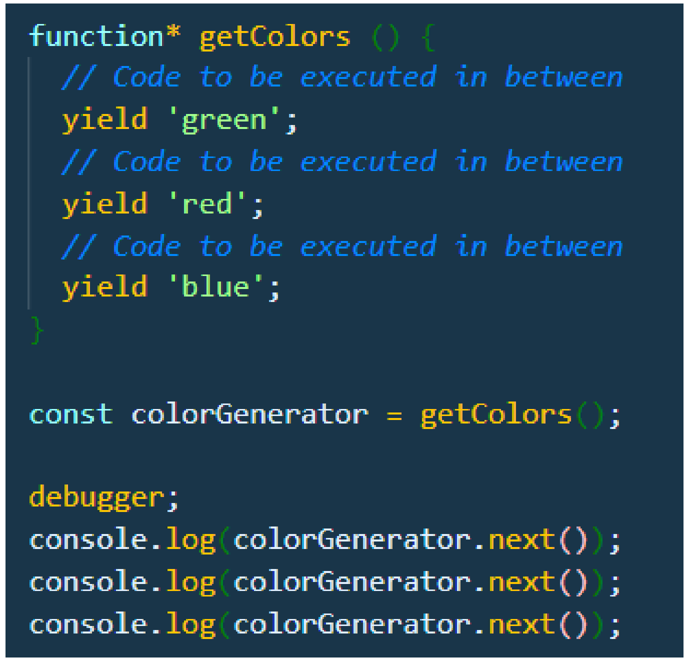
```typescript
function pickCard (x: { suit: string; card: number }[]): number;
function pickCard (x: number): { suit: string; card: number };
function pickCard (x: any): any {
  if (typeof x == 'object') {
    const pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  } else if (typeof x == 'number') {
    const pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  } else {
    return null;
  }
}
```

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



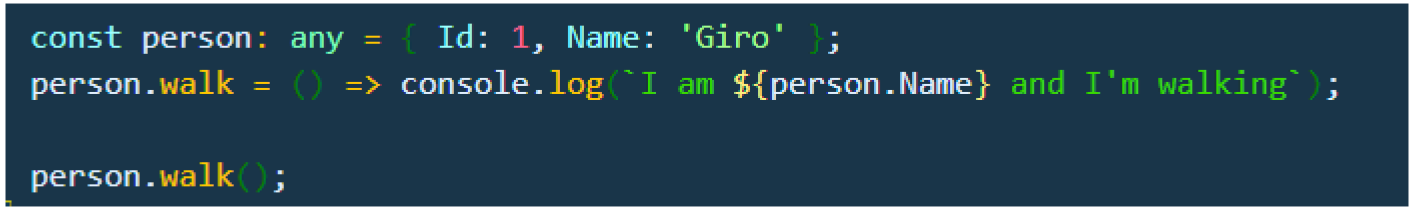
```
function* getColors () {
  // Code to be executed in between
  yield 'green';
  // Code to be executed in between
  yield 'red';
  // Code to be executed in between
  yield 'blue';
}

const colorGenerator = getColors();

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

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 {}

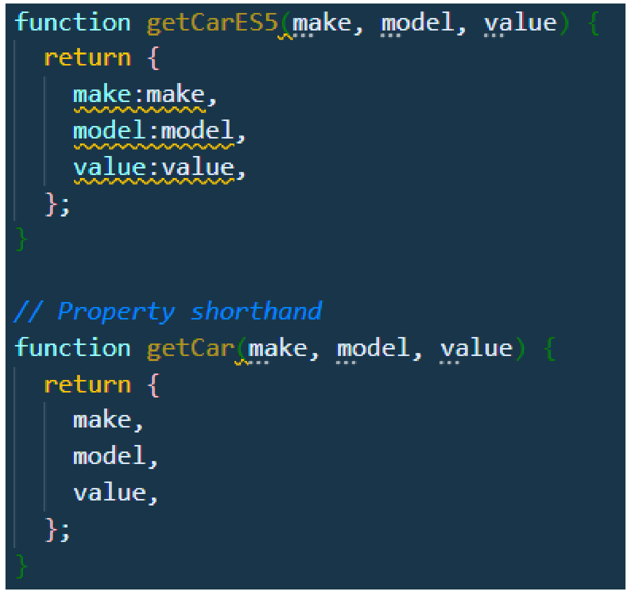
Possible to add functions to Object Literals



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

person.walk();
```

Property / Method Shorthand

New in JavaScript with ES6/ES2015, if you want to define an object who's keys have the same name as the variables passed-in as properties, you can use the shorthand and simply pass the key name



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

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

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

Immutability concepts are used in larger applications for centralized State Management e.g. using libraries like Redux (... which can be used to manage State in Angular Apps)

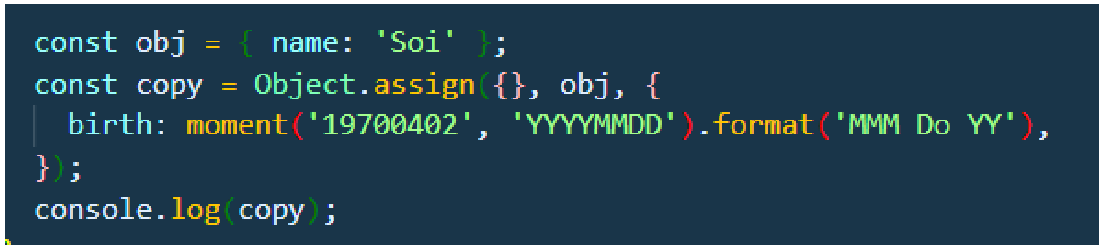
Immutability eliminates the risk of having side effects by always creating „new“ copies of objects when changing the state

object.assign()

Used to copy the values of all enumerable own properties from one or more source objects to a target object.

Used when working with architectural patterns like Redux & immutablity

Object.assign is only a copy on top level (shallow copy – not a deep copy)



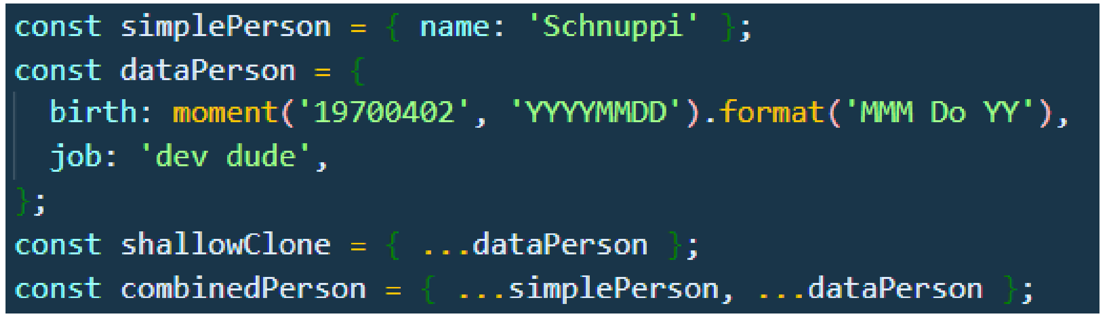
```
const obj = { name: 'Soi' };
const copy = Object.assign({}, obj, {
  birth: moment('19700402', 'YYYYMMDD').format('MMM Do YY'),
});
console.log(copy);
```

Cloning Objects using Spread Operator

Spread Operator can be used to

- create shallow copies of objects or
- combine a set of objects

Needs no polyfill for older browsers (... compared to object.assign())



```
const simplePerson = { name: 'Schnuppi' };
const dataPerson = {
  birth: moment('19700402', 'YYYYMMDD').format('MMM Do YY'),
  job: 'dev dude',
};
const shallowClone = { ...dataPerson };
const combinedPerson = { ...simplePerson, ...dataPerson };
```

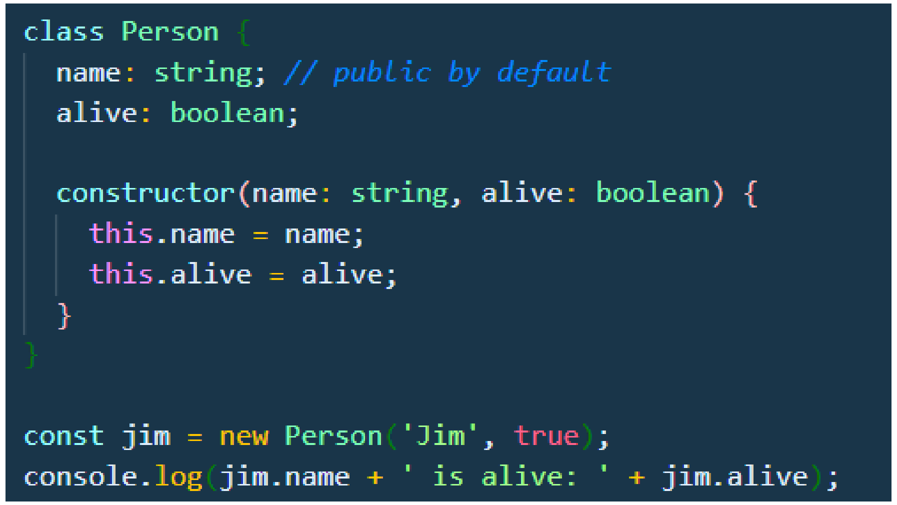
Classes, Interfaces

Classes

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

Support:

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



```
class Person {
  name: string; // public by default
  alive: boolean;

  constructor(name: string, alive: boolean) {
    this.name = name;
    this.alive = alive;
  }
}

const jim = new Person('Jim', true);
console.log(jim.name + ' is alive: ' + jim.alive);
```

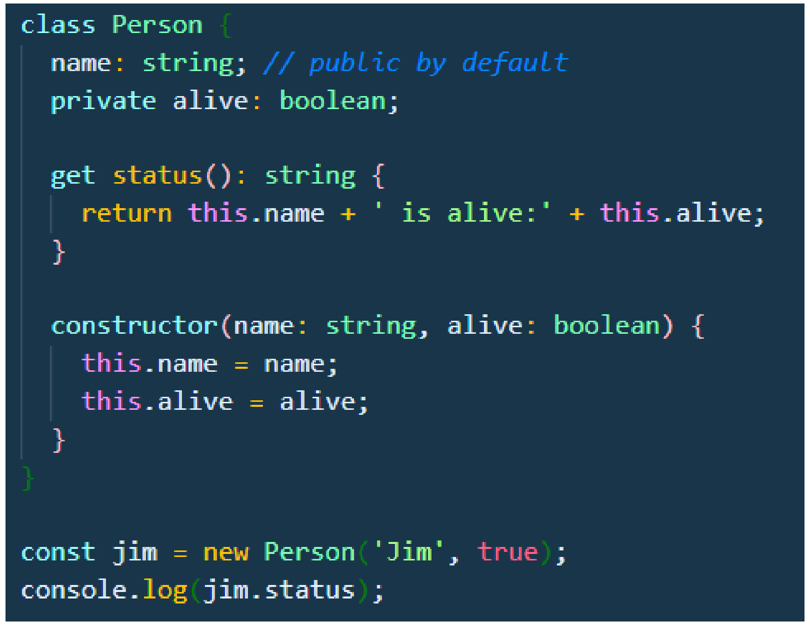
Constructor

Called when creating an instance of a class

Can be overloaded – but only with one implementation

Can define public and private properties

Can define default values and nullable parameters



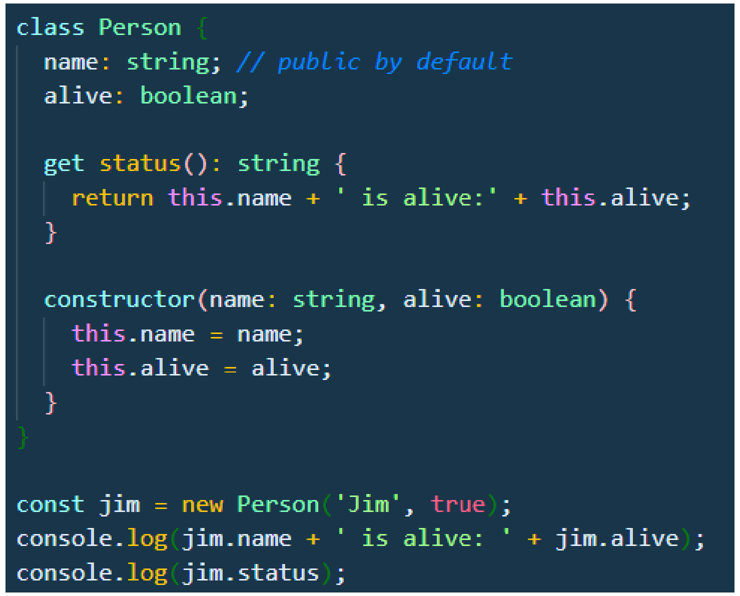
```
class Person {
  name: string; // public by default
  private alive: boolean;

  get status(): string {
    return this.name + ' is alive:' + this.alive;
  }

  constructor(name: string, alive: boolean) {
    this.name = name;
    this.alive = alive;
  }
}

const jim = new Person('Jim', true);
console.log(jim.status);
```

get / set

The get syntax binds an object property to a function that will be called when that property is looked up

The set syntax binds an object property to a function to be called when there is an attempt to set that property

- Getters are often used to combine multible props
- Setters are usually used to set private props
  - ie: Trigger next() on Stateful Services using Observables



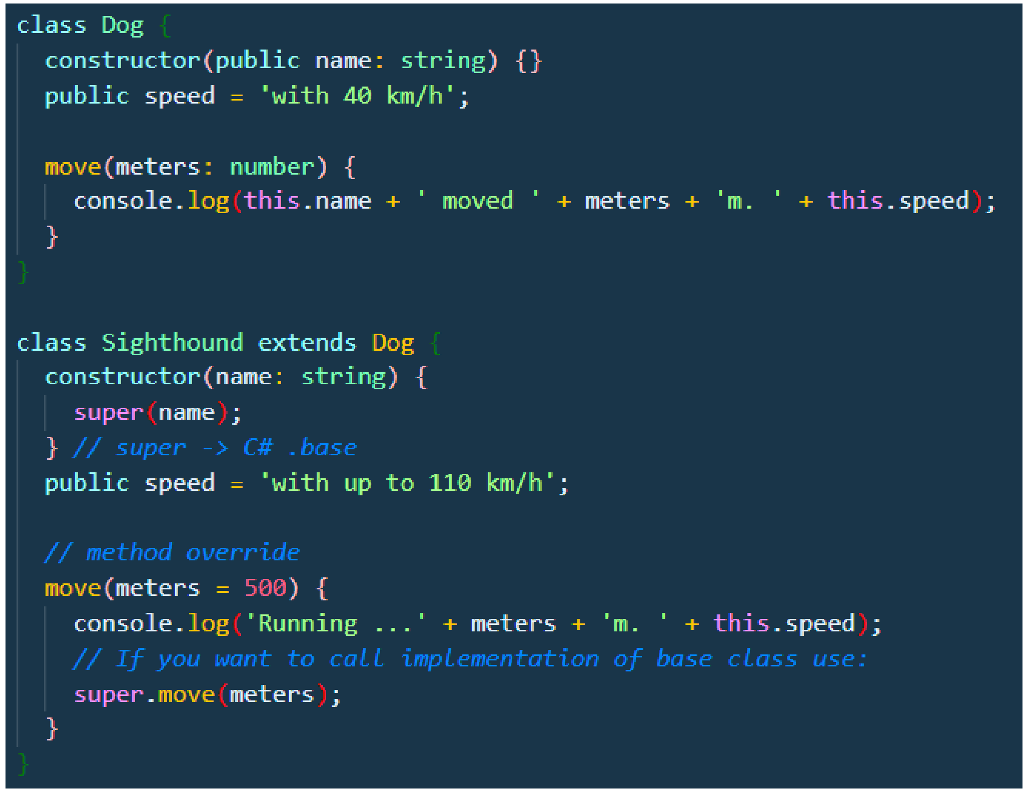
```
class Person {
  name: string; // public by default
  alive: boolean;

  get status(): string {
    return this.name + ' is alive:' + this.alive;
  }

  constructor(name: string, alive: boolean) {
    this.name = name;
    this.alive = alive;
  }
}

const jim = new Person('Jim', true);
console.log(jim.name + ' is alive: ' + jim.alive);
console.log(jim.status);
```

Class Inheritance

Class inheritance is implemented using "extends" keyword

Abstract classes are supported

Properties of the base class are accessed using "super"



```
class Dog {
  constructor(public name: string) {}
  public speed = 'with 40 km/h';

  move(meters: number) {
    console.log(this.name + ' moved ' + meters + 'm. ' + this.speed);
  }
}

class Sighthound extends Dog {
  constructor(name: string) {
    super(name);
  } // super -> C# .base
  public speed = 'with up to 110 km/h';

  // method override
  move(meters = 500) {
    console.log('Running ...' + meters + 'm. ' + this.speed);
    // If you want to call implementation of base class use:
    super.move(meters);
  }
}
```

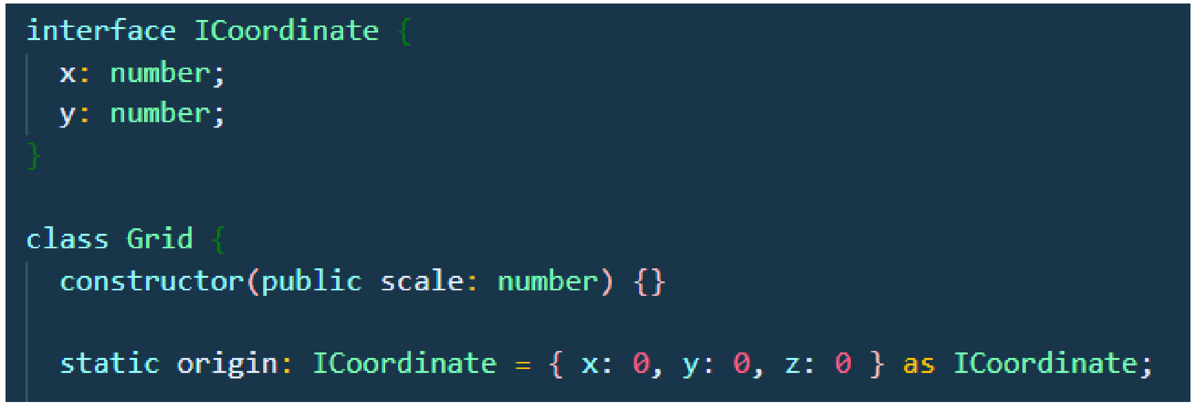
Interfaces

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

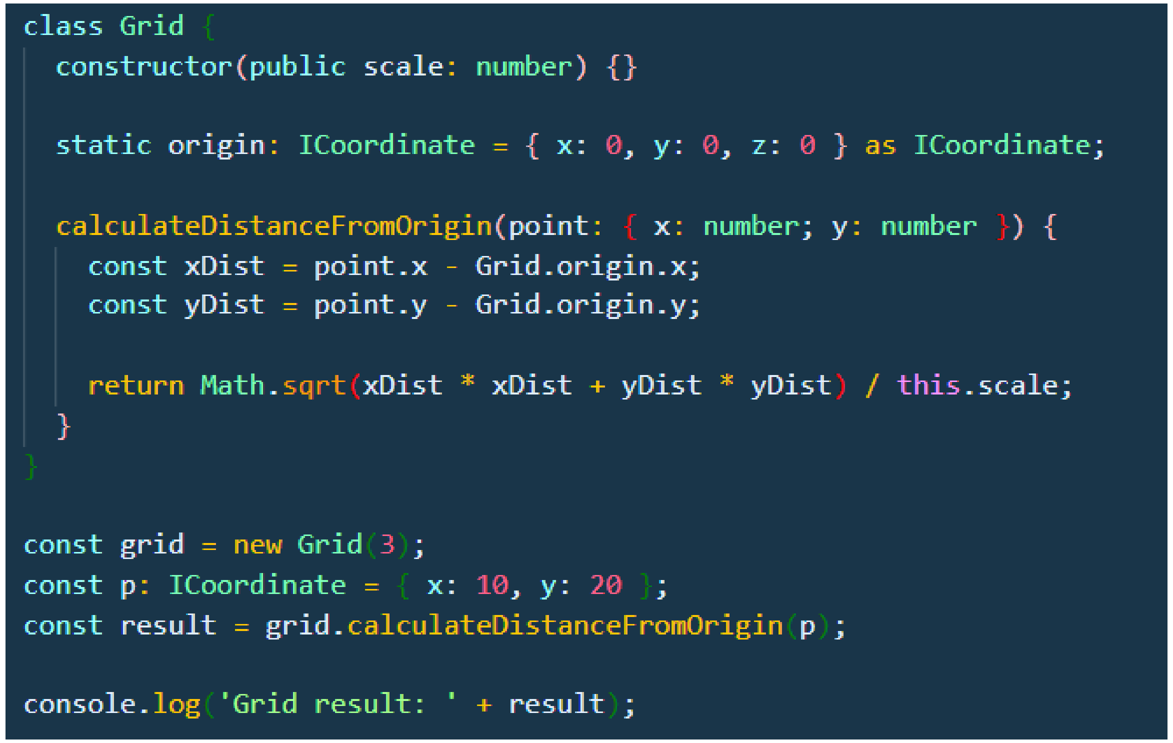


```
interface ICoordinate {
  x: number;
  y: number;
}

class Grid {
  constructor(public scale: number) {}

  static origin: ICoordinate = { x: 0, y: 0, z: 0 } as ICoordinate;
```

Static Members (Properties)

Classes in TypeScript can either contain

- static members or
- instance members.



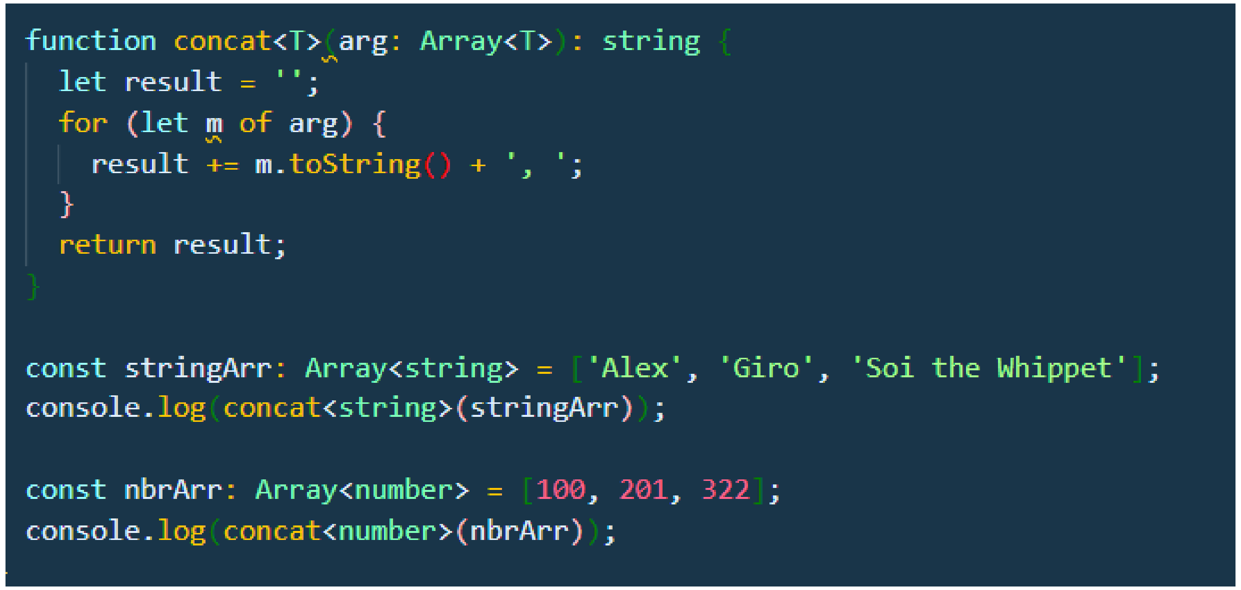
```typescript
class Grid {
  constructor(public scale: number) {}

  static origin: ICoordinate = { x: 0, y: 0, z: 0 } as ICoordinate;

  calculateDistanceFromOrigin(point: { x: number; y: number }) {
    const xDist = point.x - Grid.origin.x;
    const yDist = point.y - Grid.origin.y;

    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
}

const grid = new Grid(3);
const p: ICoordinate = { x: 10, y: 20 };
const result = grid.calculateDistanceFromOrigin(p);

console.log('Grid result: ' + result);
```

Generics, Modules, Decorators

Generic Functions

Generics allow creating reusable components that can return any given type

type is passed using <T>



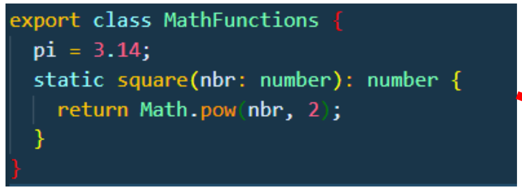
```
function concat<T>(arg: Array<T>): string {
  let result = '';
  for (let m of arg) {
    result += m.toString() + ', ';
  }
  return result;
}

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

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

Modules

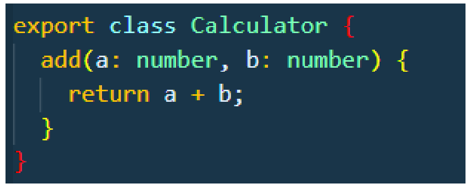
Organize other elements like classes and interfaces for better maintainability

Elements that should be visible outside must explicitly be exported / imported

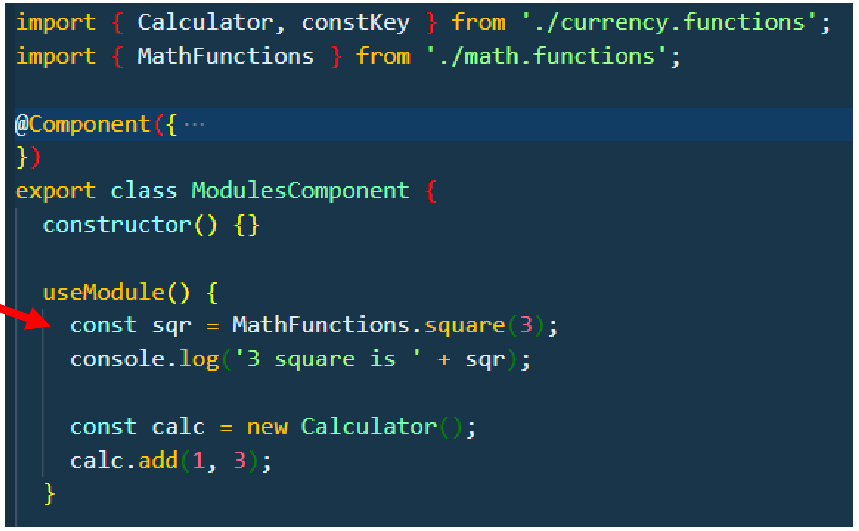
Encourage reusability



```
export class MathFunctions {
  pi = 3.14;
  static square(nbr: number): number {
    return Math.pow(nbr, 2);
  }
}
```



```
export class Calculator {
  add(a: number, b: number) {
    return a + b;
  }
}
```



```
import { Calculator, constKey } from './currency.functions';
import { MathFunctions } from './math.functions';

@Component({ ···
})
export class ModulesComponent {
  constructor() {}

  useModule() {
    const sqr = MathFunctions.square(3);
    console.log('3 square is ' + sqr);

    const calc = new Calculator();
    calc.add(1, 3);
  }
```

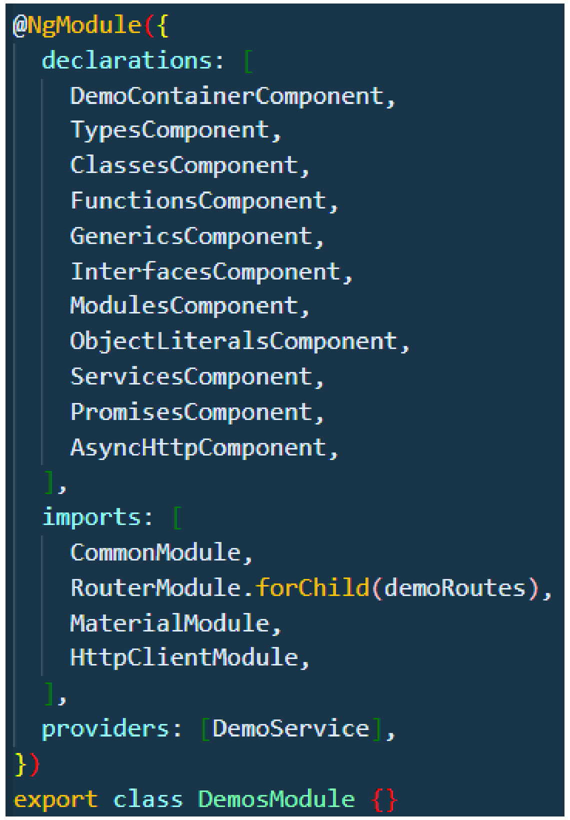
Decorators

Decorators add descriptive information to classes

- Annotations
- Metadata

Angular makes heavy use of Decoratos

- Component -> @Component
- Services -> @Injectable
- Module -> @ngModule



```
@NgModule({
  declarations: [
    DemoContainerComponent,
    TypesComponent,
    ClassesComponent,
    FunctionsComponent,
    GenericsComponent,
    InterfacesComponent,
    ModulesComponent,
    ObjectLiteralsComponent,
    ServicesComponent,
    PromisesComponent,
    AsyncHttpComponent,
  ],
  imports: [
    CommonModule,
    RouterModule.forChild(demoRoutes),
    MaterialModule,
    HttpClientModule,
  ],
  providers: [DemoService],
})
export class DemosModule {}
```

async + http requests

async + http requests

http requests calls are typically async

Possible implementation paths

- XMLHttpRequest
  - Callback based
- Fetch Api – often used together with await
  - Promise based
  - axios is an easy to use Fetch Api wrapper
- Native Clients:
  - Angular: httpClient -> Returns Observable
  - SPFx: SPHttpClient

ES 6 Promise

Pattern to deal with async - Alternative to Callbacks

- Benefit: Can be chained!

3 States: pending (in progress), fulfilled (success), rejected (error)

Use .then() & .catch() for further processing and error handling

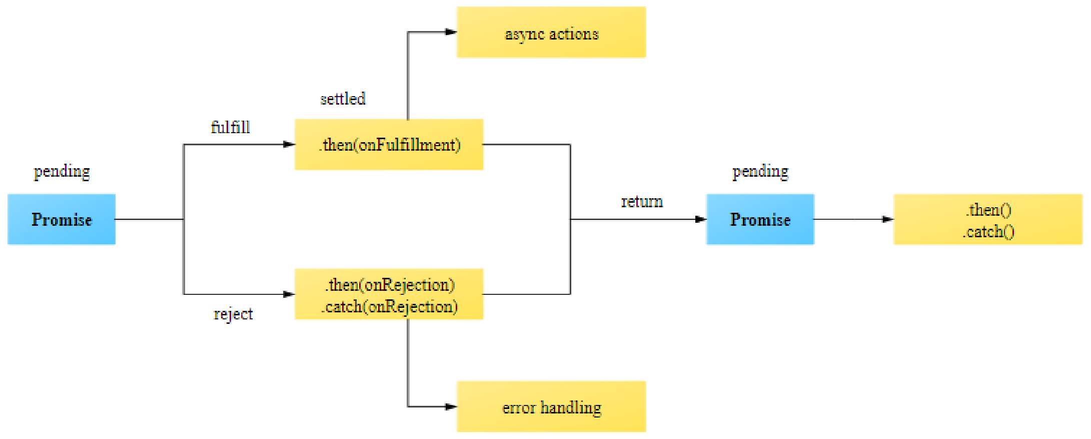
If transpiling to ES5 reference ES6-shim (https://github.com/paulmillr/es6-shim)

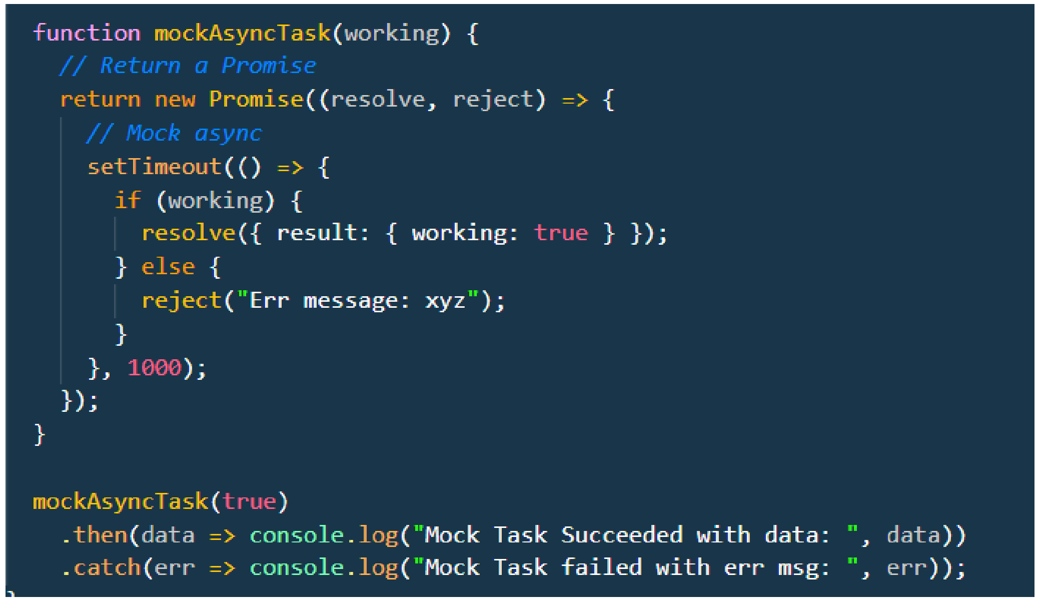
Using Promises

Use: new Promise<T>((resolve,reject))

Return Result / Err using resolve(Result) | reject(Result)

Allows Chaining!





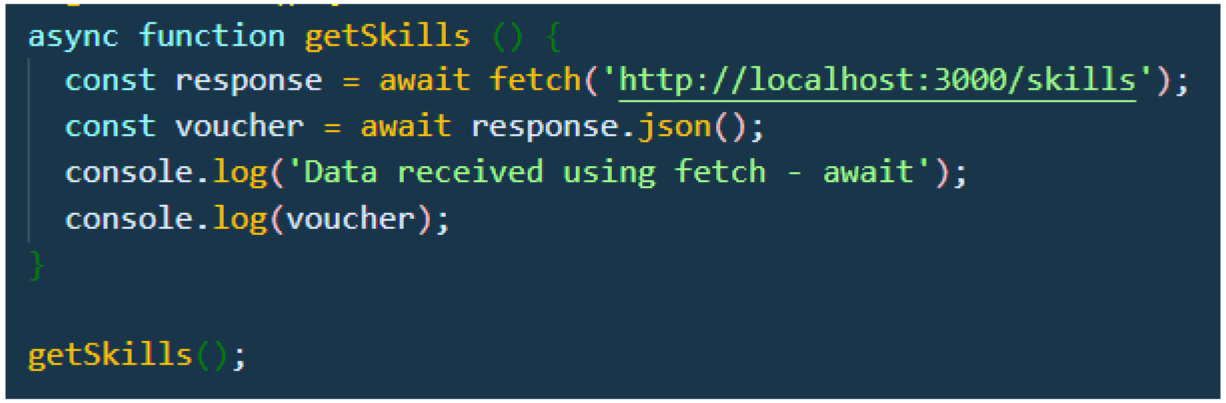
```
function mockAsyncTask(working) {
  // Return a Promise
  return new Promise((resolve, reject) => {
    // Mock async
    setTimeout(() => {
      if (working) {
        resolve({ result: { working: true } });
      } else {
        reject("Err message: xyz");
      }
    }, 1000);
  });
}

mockAsyncTask(true)
  .then(data => console.log("Mock Task Succeeded with data: ", data))
  .catch(err => console.log("Mock Task failed with err msg: ", err));
```

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



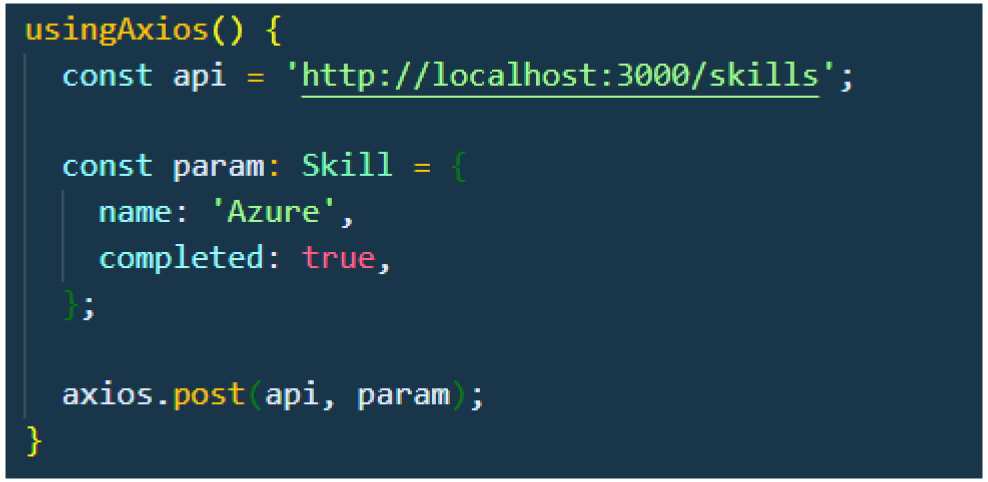
```
async function getSkills () {
  const response = await fetch('http://localhost:3000/skills');
  const voucher = await response.json();
  console.log('Data received using fetch - await');
  console.log(voucher);
}

getSkills();
```

axios

Promise based HTTP client for the browser and node.js

For people new to Promise Pattern easier to understand than Fetch API

- For other more straight to use ☺
- Of Course angular has its own httpClient!!!
- Just to Demo the Promise Pattern



```
usingAxios() {
  const api = 'http://localhost:3000/skills';

  const param: Skill = {
    name: 'Azure',
    completed: true,
  };

  axios.post(api, param);
}
```