Simple Types

BUILDING WEB APPLICATIONS USING ANGULAR

Types – just like JavaScript

- Boolean
  - true and false values only
- Number
  - Floating point values
- String
  - ‘single’ “double” or `backticks`

Types – Arrays

- Arrays
  - Statically typed arrays declared in one of two ways

```
var list:number[] = [1, 2, 3];
```

- Familiar to JavaScript developers, the square brackets

```
var list:Array<number> = [1, 2, 3];
```

- Familiar to C# and Java developers, the angle bracket notation

Types – Tuple

- Tuples
  - Express an array of fixed length but differing types

```
//Declaration
let details:[string,number];

//Initialisation
let person = [“Chris”,21] // this is fine
let person = [21, “Chris”] // error
```

Types – Enum

- Enums
  - Friendly names for numeric values
- Automatically numbered from 0

```
enum Color {Red, Green, Blue};
let c: Color = Color.Green; //1
```

- Can start from any number or number them all manually

```
enum Color {Red=4, Green, Blue};
let c: Color = Color.Green; //5
```

```
enum Color {Red=4, Green=8, Blue=16};
let c: Color = Color.Green; //8
```

- And go from numeric value to the name

```
enum Color {Red=4, Green=8, Blue=16};
let c: string = Color[8]; //Green
```

Types – Any

- All types are subtypes of Any
- Gives us a route to describe variables that we do not know the type of e.g. from the user or 3rd party libraries

```
let thing: any = “Thing T. Thing”;
thing = false //ok
```

- This is also handy to start opting in to type checking during compilation
- And can be helpful if you know parts of a type, but not all

```
let list:any[] = [1,true,”thing”];
```

Types – Void

- In some ways, the opposite of ‘any’ this type is the absence of any type at all.
- Often the type of functions that don’t return a value

```
function absenceOfThing(): void {
        alert(‘Thing has gone AWOL’);
}
```

Types – Null and Undefined

- Both primitive types in JavaScript

```
let u: undefined = undefined;
let n: null = null;
```

- They are subtypes of all other types

Types – Never

- never is a subtype of every type but nothing is a subtype of never.
- never represents the type of values that never occur:

```
//functions that never return
function notEver(): never {
        while (true) {}
}

//functions that always throw
function alwaysThrow(): never {
        throw new Error(“throwing”);
}
```

Types – Type Assertion

- Sometimes as developers we need to override the compiler
  - Usually when an entity is more specific than its current type
- TypeScript provides two syntaxes

```
//angle-bracket syntax
let thing: any = “Thing T. Thing”;
let nameLength: number = (<string>thing).length;

//as-syntax
let thing: any = “Thing T. Thing”;
let nameLength: number = (thing as string).length;
```

Classes

BUILDING WEB APPLICATIONS USING ANGULAR

Classes

- Syntactic sugar over prototypal inheritance
- Gotcha: NOT hoisted like functions
- Executed in strict mode

```
class Car {
  wheels: number;
  power: number;
  speed: number = 0;

  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }

  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
let myCar = new Car(4, 20); //constructor called
```

Classes: Extends

- The extends and super keywords allow sub-classing

```
class Vehicle {
  wheels: number;
  power: number;
  speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
}
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) {
    super(wheels, power); //call the parent constructor
    this.gps = true; //GPS as standard
  }
}

let myCar = new Car(4, 20);
```

Classes: Modifiers (Public, Private, Protected)

- Public, Private and Protected
- Public is the default behavior but can be specified

```
class Car {
  public wheels: number;
  public power: number;
  public speed: number = 0;
  public constructor (wheels, power) {
    this.wheels = wheels;
    this.power = power;
}

  public accelerate(time) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

let myCar = new Car(4, 20); //constructor called
```

Classes: Modifiers (Public, Private, Protected)

- Private modifier prevents access from outside the class

```
class Car {
  private wheels: number;
  private power: number;
  private speed: number = 0;
  constructor (wheels, power) {
    this.wheels = wheels;
    this.power = power;
  }

  public accelerate(time) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

let myCar = new Car(4, 20);
console.log(myCar.speed); //error ‘speed’ is private
```

Classes: Modifiers (Public, Private, Protected)

- Protected modifier acts much like private except protected members can be accessed by deriving classes.

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  protected speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) { super(wheels, power); }
  public showSpeed() {
    return `Current speed: ${this.speed}`
  }
}

let myCar = new Car(4, 20);
console.log(myCar.showSpeed());
console.log(myCar.speed); //error
```

Classes: Modifiers (Public, Private, Protected)

- We can protect constructors to enable extension but not instantiation

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  protected speed: number = 0;
  protected constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) { super(wheels, power); }
  public showSpeed() {
    return `Current speed: ${this.speed}`
  }
}

let myCar = new Car(4, 20);
let myVehicle = new Vehicle(4,20); //Error constructor is protected
```

Classes: Structural Types

- TypeScript is a structural type system – if the types of all members ae compatible, then the types are compatible.
- Except for private and protected members.

Classes: Structural Types

```
class Vehicle {
  public wheels: number;
  public power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
}

class RCCar {
  public wheels: number;
  public power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

let myCar = new Car(4, 20);
let myRCCar = new RCCar(4,5);
let vehicle = new Vehicle(4,15);

vehicle = myCar; //ok
vehicle = myRCCar; //ok
```

Classes: Structural Types

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
}

class RCCar {
  protected wheels: number;
  protected power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

let myCar = new Car(4, 20);
let myRCCar = new RCCar(4,5);
let vehicle = new Vehicle(4,15);

vehicle = myCar; //ok
vehicle = myRCCar; //Error: RCCar is not a subclass of Vehicle
```

Classes: Readonly

- Readonly properties must be initialised at their decleration or in the constructor

```
class Vehicle {
  readonly wheels: number;
  readonly power: number;
  protected speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
}
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  readonly gps: Boolean = true;
  constructor (wheels, power) {
    super(wheels, power);
  }
}

let myCar = new Car(4, 20);
myCar.wheels = 3; //error - readonly property
```

Classes: Parameter Properties

- Parameter properties stop us repeating ourselves quite so much by creating and initialising a property in one place

```
class Vehicle {
  protected speed: number = 0;
  constructor (readonly wheels: number, readonly power: number) {
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  readonly gps: Boolean = true;
  constructor (wheels, power) {
    super(wheels, power);
  }
}

let myCar = new Car(4, 20);
console.log(myCar.wheels); //4
```

- By using a modifier in the parameter we create a property

Classes: Getters and Setters

- Changing properties directly can often be a bad idea, leading to tightly coupled code
- Getters and Setters allow us to:
  - Encapsulate our implementation
  - Add logic to properties

Classes: Getters and Setters

```
class Car {
  private _speed: number = 0;
  constructor (readonly wheels: number, readonly power: number) {
  }

  get speed(): number {
    return this._speed;
  }

  set speed(newSpeed: number) {
    if (newSpeed && newSpeed > -30 && newSpeed <= 150) {
      this._speed = newSpeed;
    }
  }

  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

let myCar = new Car(4, 20);
console.log(myCar.speed) //0
myCar.speed = 100;
console.log(myCar.speed) //100
myCar.speed = 151;
console.log(myCar.speed) //100
myCar._speed = 151 // Error
```

Static Properties

- We can create static members that are visible on the class itself rather than its instances
- Useful for data and behaviour that does not change depending on instance

```
class Car {
  private speed: number = 0;
  static count: number = 0;
  constructor (readonly wheels: number, readonly power: number) {
    Car.count += 1;
  }



  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

for (let i = 0; i < 10; i++) {
  new Car(4,20);
}

console.log(Car.count); //10
```

Abstract Classes

- Abstract classes allow us to create base classes from which other classes may be derived.
- Abstract classes cannot be instantiated themselves.
- Abstract classes provide implementation details

```
abstract class Vehicle {
  wheels: number;
  power: number;
  speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  abstract accelerate(time: number): void;
}

class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
  public accelerate(time: number): void {
     this.speed = this.speed + 0.5*this.power*time;
  }
}

let myCar = new Car(4, 20);
myCar.accelerate(5);
let myVehicle = new Vehicle(4,20); //Error
```

Modules

BUILDING WEB APPLICATIONS USING ANGULAR

Modules - Nomenclature

- Pre typescript 1.5 there was a concept of “internal modules” and “external modules”
- ECAMScript2015 introduced “modules” to JavaScript and so TypeScript has changed its terminology to match.
- Internal modules are now “namespaces”
- External modules are now simply “modules”

Modules

- Modules run in their own scope, avoiding pollution of the global scope
- Only what is exported is exposed externally
- Only what is imported is usable internally
- Any declaration can be exported through using the export keyword

```
//vehicles.ts
export interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}

export class Car implements Vehicle {
  wheels = 4;
  constructor(public make:string, public model:string){}
  accelerate(time: number) {…}
}
```

Modules: Export Statements

- Export statements can be used to export under a different name

```
//vehicles.ts
interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}

class Car implements Vehicle {
  wheels = 4;
  constructor(public make:string, public model:string){}
  accelerate(time: number) {…}
}

export { Vehicle };
export { Car as BaseCar };
```

Modules: Default Export

- Optionally a default export can be specified

```
//vehicles.ts
export default interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}

class Car implements Vehicle {
  wheels = 4;
  constructor(public make:string, public model:string){}
  accelerate(time: number) {…}
}

//alternative default export syntax
//note: can not have two default export statements
export { Car as default };
```

Modules: Importing

- Importing is just as simple as exporting. We use the import keyword with one of the following forms

```
import { Vehicle, Car } from ‘./vehicles’;

let myCar = new Car(‘Ford’,’Fiesta’);
```

- We can rename on import

```
import { Car as BasicCar} from ‘./vehicles’;

let myCar = new BasicCar(‘Ford’,’Fiesta’);
```

- Or import the whole file!

```
import * as vehicles from ‘./vehicles’;

let myCar = new vehicles. Car(‘Ford’,’Fiesta’);
```

- Importing the default is the simplest form

```
import Car from ‘./vehicles’
```

Modules: export = and import = require()

- The two most common module syntaxes prior to ES2015 (CommonJS and AMD) both supported the concept of an exports object. TypeScript has it’s own syntax to model this workflow.

```
//vehicles.ts
class Car implements Vehicle {
  wheels = 4;
  constructor(public make:string, public model:string){}
  accelerate(time: number) {…}
}

export = Car;
```

```
//app.ts
import Car = require(‘./vehicles’);
```

Modules: Code Creation

- The TypeScript compiler is not a module loader. It will compile your code to whatever module format you tell it to.
- A module loader will be required to then make your code ready for deployment

Decorators

BUILDING WEB APPLICATIONS USING ANGULAR

Introduction

- Decorators provide a means through which existing classes and class members can be annotated and modified
- They are an experimental feature for JavaScript that are available now in TypeScript – so they may change in future releases!
- Decorators use a special @expression syntax where the expression evaluates to a function which takes information about the decorated declaration

```
//@myDecorator

function myDecorator(target: any) {
    //do some stuff with target
}
```

My First Decorator

- Where you can use a decorator is dependent on the parameters you supply the function with

```
//A simple class decorator

function simpleDecorator(target: any) {
    console.log(‘My first decorator was called’)
}

@simpleDecorator
class DecoratedClass {

}
```

- Our (class) decorator takes the constructor as its only argument and simply logs the message to the console
- We don’t need to instantiate the class for the decorator to run!

Decorator Factories

- We can use decorator factories to be able to provide our decerators with parameters
- Remember: a Decorator should evaluate to a function

```
function DecoratorFactory(name: string) {
    return function(target: Function) {
        console.log(`${name} decorator was called`)
    }
}

@DecoratorFactory("factory")
    class DecoratedClass {
}
```

Class Decorator Parameter

- The runtime automatically passes the parameters to the evaluated function of our Decorators
- In the case of a Class decorator this is the constructor function itself

```
function merge(toMerge: Object) {
    return function (target: any) {
        for (let prop in toMerge) {
            target.prototype[prop] = toMerge[prop];
        }
    }
}

let user = {
    name: 'Chris Bruford',
    age: 22,
    instructor: true
}

@merge(user)
class DecoratedClass {
    constructor() {};
    test = true;
}

let thing = new DecoratedClass();
console.log((<any>thing).name);  //cast to ‘any’ in order to use the name property
```

Method Decorators

- Remember we can decorate any class or class member
- In the case of a method decorator the arguments required for the decorator are:
  - The target – the class prototype
  - The method name
  - The method descriptor

```
function readOnly(target: any, methodName: string, descriptor?: PropertyDescriptor) {
    descriptor.writable = false;
    descriptor.enumerable = false;
}

class DecoratedClass {
    @readOnly
    sayHello() { console.log("Hello") }
}

let thing = new DecoratedClass();
thing.sayHello()
thing.sayHello = false; //error (in strict mode – silent fail otherwise)
```