Classes

- Classes are heavily used in classical object oriented programming
- It defines what an object is and what it can do
- A class is defined using the `class` keyword followed by a name
- By convention, the name of the class start with an uppercase letter
- A class can be used to create multiple objects (instances) of the same class
- An object is created from a class using the `new` keyword
- A class can have a `constructor` which is called when an object is made from the class
- Properties of a class are called instance variables and its functions are called the class methods
- Access modifiers can be used to make them public or private
- The instance variables are attached to the instance itself but not the prototype
- Methods however are attached to the prototype object as opposed to the instance itself
- Classes can inherit functionality from other classes, but you should favor composition over inheritance or make sure you know when to use it
- Classes can implement interfaces

Let's make a class definition for a car and incrementally add more things to it. The project files for this section are in `angular2-intro/project-files/typescript/classes/basic-class`.

Adding an Instance Variable

The `Car` class definition can be very simple and can define only a single instance variable that all cars can have:

```
class Car {
  distance: number;
}
```

- `Car` is the name of the class, which also defines the custom type `Car`
- `distance` is a property that tracks the distance that car has traveled
- Distance is of type `number` and only accepts `number` type.

Now that we have the definition for a car, we can create a car from the definition:

```
let myCar:Car = new Car();
myCar.distance = 0;
```

- `myCar:Car` means that `myCar` is of type `Car`
- `new Car()` creates an instance from the `Car` definition.
- `myCar.distance = 0` sets the initial value of the `distance` to 0 for the newly created `car`

Adding a Method

So far our car doesn't have any definitions for any actions. Let's define a `move` method that all the cars can have:

```
class Car {
  distance: number;
  move():void {
    this.distance += 1;
  }
}
```

- `move():void` means that `move` is a method that does not return any value, hence `void`.
- The body of the method is defined in `{ }`
- `this` refers to the instance, therefore `this.distance` points to the `distance` property defined on the car instance.
- Now you can call the `move` method on the car instance to increment the `distance` value by 1:

```
myCar.move();
console.log(myCar.distance) // -> 1
```

Using Access Modifiers

If you wanted to tell the compiler that the `distance` variable is private and can only be used by the object itself, you can use the `private` modifier before the name of the property:

```
class Car {
  private distance: number;
  constructor () {
    ...
  }
  ...
}
```

- There are 3 main access modifiers in TypeScript: `private`, `public`, and `protected`:

- `private` modifier means that the property or the method is only defined inside the class only.
- `protected` modifier means that the property or the method is only accessible inside the class and the classes derived from the class.
- `public` is the default modifier which means the property or the method is the accessible everywhere and can be accessed by anyone.

Adding a constructor

A `constructor` is a special method that gets called when an instance is created from a class. A class may contain at most one constructor declaration. If a class contains no constructor declaration, an automatic constructor is provided.

Let's add a constructor to the `Car` class that initializes the `distance` value to 0. This means that all the cars that are crated from this class, will have their `distance` set to 0 automatically:

```
class Car {
  distance: number;
  constructor () {
    this.distance = 0;
  }
  move():void {
    this.distance += 1;
  }
}
```

- `constructor()` is called automatically when a new car is created
- Parameters are passed to the constructor in the `()`
- The body of the constructor is defined in the `{ }`

Now, let's customize the car's constructor to accept `distance` as a parameter:

```
1  class Car {
2    private distance: number;
3    constructor (distance) {
4      this.distance = distance;
5    }
6  }
```

- On line 3 we are passing distance as a parameter. This means that when a new instance is created, a value should be passed in to set the distance of the car.
- On line 4 we are assigning the value of distance to the value that is passed in

This pattern is so common that TypeScript has a shorthand for it:

```
class Car {
  constructor (private distance) {
  }
}
```

Note that the only thing that we had to do was to add `private distance` in the constructor parameter and remove the `this.distance` and `distance: number`. TypeScript will automatically generate that. Below is the JavaScript outputed by TypeScript:

```
var Car = (function () {
  function Car(distance) {
    this.distance = distance;
  }
  return Car;
})();
```

Now that our car expects a `distance` we have to always supply a value for the distance when creating a car. You can define default values if you want so that the car is instantiated with a default value for the distance if none is given:

```
class Car {
  constructor (private distance = 0) {
  }
  getDistance():number { return this.distance; }
}
```

Now if I forget to pass a value for the `distance`, it is going to be set to zero by default:

```
const mycar = new Car();
console.log(mycar.getDistance()); //-> 0
```

Note that if you pass a value, it will override the default value:

```
const mycar = new Car(5);
console.log(mycar.getDistance()); //-> 5
```

Setters and Getters (Accessors)

It is a very common pattern to have setters and getters for properties of a class. TypeScript provides a very simple syntax to achieve that. Let's take our example above and add a setter and getter for the distance property. But before that we are going to rename `distance` to `_distance` to make it explicit that it is private. It is not required but it is a common pattern to prefix private properties with an underscore.

```
1  class Car {
2    constructor (private _distance = 0) {}
3    getDistance():number { return this._distance; }
4  }
```

In order to create the getter method, we are going to use the `get` keyword and the name for the property followed by `()` :

```
1  class Car {
2    constructor (private _distance = 0) {}
3    get distance() { return this._distance; }
4  }
```

Now we can get the value of `distance` :

```
1  const car2 = new Car(5);
2  console.log(car2.distance) //-> 5
```

Note on line 2 that we didn't call a function. Behind the scenes, TypeScript creates a property for us, that's why it is not a method. Below is the relevant generated JavaScript:

```
1  Object.defineProperty(Car.prototype, "distance", {
2    get: function () { return this._distance; },
3    enumerable: true,
4    configurable: true
5  });
```

JavaScript behind the scenes calls the get function for you to get the value, and that's why we simply did `car2.distance` as opposed to `car2.distance()`. For more information about `Object.defineProperty` checkout the MDN docs.

Similar to the getter, we can define a setter as well:

```
class Car {
  constructor (private _distance = 0) {}
  get distance() { return this._distance; }
  set distance(newDistance: number) { this._distance = newDistance; }
}
```

Now we can both get and set the distance value:

```
const coolCar = new Car();
console.log(coolCar.distance); // -> 0

coolCar.distance = 55;
console.log(coolCar.distance); // -> 55
```

Note that if we take out the setter, we won't be able to assign a new value to `distance`.

Static Methods and Properties

Static methods and properties belong to the class but not the instances. For example, the `Array.isArray` method is only accessible through the `Array` but not an instance of an array:

```
1 var x = [];
2 x.isArray // -> undefined
3 Array.isArray(x) //-> true
```

- On line 2 we are trying to access the `isArray` method, but obviously it is not defined because `isArray` is a static method.

- On line three we are calling the static `isArray` method from `Array` and we can check if `x` is an array.

If you look at the Array documentation you can see that methods and properties are either defined on the `Array.prototype` or `Array`:

- `Array.prototype.x`: makes `x` available to all the instances of `Array`
- `Array.x`: `x` is static and only available through `Array`.

Now that we have some context, let's see how you can define static methods and properties in TypeScript. Consider the code below:

```
1  class Car {
2    static controls: {isAuto: boolean } = {
3      isAuto: true
4    };
5    static isAuto():boolean {
6      return Car.controls.isAuto;
7    }
8    constructor (private _distance = 0) {}
9    get distance() { return this._distance; }
10 }
11
12 console.log(Car.controls); // -> { isAuto: true }
13 console.log(Car.isAuto()); // -> true
```

- On line 2 we are defining a static property called `controls` using the `static` modifier. Then we specify the form and then assign a value for it.
- On line 5 we are defining a static method called `isAuto` using the the `static` modifier. This method simply returns the value of `isAuto` from the static `control` object. Not that we get access to the class using the name of the class as opposed to using `this`. i.e. `return Car.controls.isAuto`

Implementing an Interface

Classes can implement one or multiple interfaces. We can make the `Car` class implement two interfaces:

```typescript
interface ICarProps {
  distance: number;
}
interface ICarMethods {
  move():void;
}
```

Making the `Car` class implement the interfaces:

```typescript
class Car implements ICarProps, ICarMethods {
  distance: number;
  constructor () {
    this.distance = 5;
  };
  move():void {
    this.distance += 1;
  };
}
```

The above example is silly, but it shows the point that a class can implement one or more interfaces. Now if the class does not provide implementations for any of the interfaces, the compiler will complain. For example, if we leave out the `distance` instance variable, the compiler will print out the following error:

> error TS2420: Class 'Car' incorrectly implements interface 'ICarProps'. Property 'distance' is missing in type 'Car'.

Inheritance

In Object-oriented programming, a class can inherit from another class which helps to define shared attributes and methods among objects. Although this pattern is very useful, it should be used cautiously as it can lead to code that is hard to maintain. You can learn more about classical inheritance and prototypical inheritance by watching Eric Elliot's talk at O'Reilly's

Fluent Conference. The project files for this section are in `angular2-intro/project-files/typescript/classes/inheritance` .

Let's get started by creating a base class called `Vehicle` . This class is going to be the base class for other classes that we create later.

```
// Vehicle.ts
export class Vehicle {
  constructor( private _name: string = 'Vehicle',
               private _distance: number = 0 ) { }
  get distance(): number { return this._distance; }
  set distance(newDistance: number) { this._distance = newDistance; }
  get name(): string { return this._name;}
  set name(newName: string) { this._name = newName; }
  move() { this.distance += 1 }
  toString() { return this._name; }
}
```

There is nothing special in this class. We are just creating a class that has two private properties (name, distance) and we are creating the setters and getters for them. Additionally, we are defining the `toString` method that JavaScript internally calls in "textual contexts". The constructor is the most notable of all the other methods:

- It sets the `name` property to "Vehicle" for all the instances
- It also sets the `distance` property to 0.

This means that when a class extends the `Vehicle` class, it will have to call the constructor of `Vehicle` using the `super` keyword. Let's do that now by creating two classes called `Car` and `Truck` that inherit from the `Vehicle` class:

`cars.ts`

```
import {Vehicle} from './vehicle';
export class Car extends Vehicle {
  constructor(name?: string) {
    super();
    this.name = name || 'Car';
  }
}
export class Truck extends Vehicle {
  constructor(name?: string) {
    super();
    this.name = name || 'Truck';
  }
}
```

- The `Car` class and the `Truck` class both look almost identical. They both inherit from the `Vehicle` using the `extends` keyword.
- They both call the `Vehicle`'s constructor in their own constructor method before implementing their own: `constructor(name?: string) { super(); }`
- They both take an optional `name` property to set the name of the vehicle. If not name is provided, it will be set to either 'Car' or 'Truck'

Now let's create the `main` file and run the file:

```
1  import {Car, Truck} from './cars';
2  
3  /**
4   * Creating a new car from `Car`
5   */
6  const car = new Car();
7  console.log(car.name);
8  car.distance = 5;
9  car.move();
10 car.move();
11 console.log(car.distance);
12 /**
13  * Creating a new Truck.
14  */
15 const truck = new Truck();
16 console.log(truck.name);
```

- On line 1 we are importing the `Car` and the `Truck` class.

- and then we create a `Car` and `Truck` instance and log their names and distance to the console.

Run the build task (command + shift + b) and run the file (F5) and you should see the output:

```
node --debug-brk=7394 --nolazy output/main.js
Debugger listening on port 7394
Car
7
Truck
```

You can play around with the code above an try passing a string when instantiating a `Car` or a `Truck` to see the name change.

TODO

- `constructor overloading`