# Chapter 7: Classes

TypeScript, like ECMAScript 6, support object-oriented programming using classes. This contrasts with older JavaScript versions, which only supported prototype-based inheritance chain.

The class support in TypeScript is similar to that of languages like Java and C#, in that classes may inherit from other classes, while objects are instantiated as class instances.

Also similar to those languages, TypeScript classes may implement interfaces or make use of generics.

## Section 7.1: Abstract Classes

```
abstract class Machine {
    constructor(public manufacturer: string) {
    }

    // An abstract class can define methods of its own, or...
    summary(): string {
        return `${this.manufacturer} makes this machine.`;
    }

    // Require inheriting classes to implement methods
    abstract moreInfo(): string;
}

class Car extends Machine {
    constructor(manufacturer: string, public position: number, protected speed: number) {
        super(manufacturer);
    }

    move() {
        this.position += this.speed;
    }

    moreInfo() {
        return `This is a car located at ${this.position} and going ${this.speed}mph!`;
    }
}

let myCar = new Car("Konda", 10, 70);
myCar.move(); // position is now 80
console.log(myCar.summary()); // prints "Konda makes this machine."
console.log(myCar.moreInfo()); // prints "This is a car located at 80 and going 70mph!"
```

Abstract classes are base classes from which other classes can extend. They cannot be instantiated themselves (i.e. you **cannot** do `new Machine("Konda")`).

The two key characteristics of an abstract class in TypeScript are:

1. They can implement methods of their own.
2. They can define methods that inheriting classes **must** implement.

For this reason, abstract classes can conceptually be considered a **combination of an interface and a class**.

## Section 7.2: Simple class

```
class Car {
```

```
    public position: number = 0;
    private speed: number = 42;

    move() {
        this.position += this.speed;
    }
}
```

In this example, we declare a simple class `Car`. The class has three members: a private property `speed`, a public property `position` and a public method `move`. Note that each member is public by default. That's why `move()` is public, even if we didn't use the `public` keyword.

```
var car = new Car();        // create an instance of Car
car.move();                 // call a method
console.log(car.position);  // access a public property
```

## Section 7.3: Basic Inheritance

```
class Car {
    public position: number = 0;
    protected speed: number = 42;

    move() {
        this.position += this.speed;
    }
}

class SelfDrivingCar extends Car {

    move() {
        // start moving around :-)
        super.move();
        super.move();
    }
}
```

This examples shows how to create a very simple subclass of the `Car` class using the `extends` keyword. The `SelfDrivingCar` class overrides the `move()` method and uses the base class implementation using `super`.

## Section 7.4: Constructors

In this example we use the `constructor` to declare a public property `position` and a protected property `speed` in the base class. These properties are called *Parameter properties*. They let us declare a constructor parameter and a member in one place.

One of the best things in TypeScript, is automatic assignment of constructor parameters to the relevant property.

```
class Car {
    public position: number;
    protected speed: number;

    constructor(position: number, speed: number) {
        this.position = position;
        this.speed = speed;
    }

    move() {
        this.position += this.speed;
```

```
    }
}
```

All this code can be resumed in one single constructor:

```
class Car {
    constructor(public position: number, protected speed: number) {}

    move() {
        this.position += this.speed;
    }
}
```

And both of them will be transpiled from TypeScript (design time and compile time) to JavaScript with same result, but writing significantly less code:

```
var Car = (function () {
    function Car(position, speed) {
        this.position = position;
        this.speed = speed;
    }
    Car.prototype.move = function () {
        this.position += this.speed;
    };
    return Car;
}());
```

Constructors of derived classes have to call the base class constructor with `super()`.

```
class SelfDrivingCar extends Car {
    constructor(startAutoPilot: boolean) {
        super(0, 42);
        if (startAutoPilot) {
            this.move();
        }
    }
}

let car = new SelfDrivingCar(true);
console.log(car.position);  // access the public property position
```

# Section 7.5: Accessors

In this example, we modify the "Simple class" example to allow access to the `speed` property. TypeScript accessors allow us to add additional code in getters or setters.

```
class Car {
    public position: number = 0;
    private _speed: number = 42;
    private _MAX_SPEED = 100

    move() {
        this.position += this._speed;
    }

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

```
    set speed(value: number) {
        this._speed = Math.min(value, this._MAX_SPEED);
    }
}

let car = new Car();
car.speed = 120;
console.log(car.speed);  // 100
```

# Section 7.6: Transpilation

Given a class `SomeClass`, let's see how the TypeScript is transpiled into JavaScript.

**TypeScript source**

```
class SomeClass {

    public static SomeStaticValue: string = "hello";
    public someMemberValue: number = 15;
    private somePrivateValue: boolean = false;

    constructor () {
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }

    public static getGoodbye(): string {
        return "goodbye!";
    }

    public getFortyTwo(): number {
        return 42;
    }

    private getTrue(): boolean {
        return true;
    }

}
```

**JavaScript source**

When transpiled using TypeScript `v2.2.2`, the output is like so:

```
var SomeClass = (function () {
    function SomeClass() {
        this.someMemberValue = 15;
        this.somePrivateValue = false;
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }
    SomeClass.getGoodbye = function () {
        return "goodbye!";
    };
    SomeClass.prototype.getFortyTwo = function () {
        return 42;
    };
    SomeClass.prototype.getTrue = function () {
        return true;
    };
```

```
    return SomeClass;
}());
SomeClass.SomeStaticValue = "hello";
```

**Observations**

- The modification of the class' prototype is wrapped inside an IIFE.
- Member variables are defined inside the main class `function`.
- Static properties are added directly to the class object, whereas instance properties are added to the prototype.

## Section 7.7: Monkey patch a function into an existing class

Sometimes it's useful to be able to extend a class with new functions. For example let's suppose that a string should be converted to a camel case string. So we need to tell TypeScript, that `String` contains a function called `toCamelCase`, which returns a `string`.

```
interface String {
    toCamelCase(): string;
}
```

Now we can patch this function into the `String` implementation.

```
String.prototype.toCamelCase = function() : string {
    return this.replace(/[^a-z ]/ig, '')
        .replace(/(?:^\w|[A-Z]|\b\w|\s+)/g, (match: any, index: number) => {
            return +match === 0 ? "" : match[index === 0 ? 'toLowerCase' : 'toUpperCase']();
        });
}
```

If this extension of `String` is loaded, it's usable like this:

```
"This is an example".toCamelCase();    // => "thisIsAnExample"
```