TypeScript Classes

Classes of TypeScript

Motivation

With some more understanding of typescript, we will now look into classes.

Classes - JavaScript

In JavaScript: Classes are a template for creating objects. They encapsulate data with code to work on that data.

Classes in JS are built **`on prototypes`** but also have some syntax and semantics that are not shared with ES5 class-like semantics.

`Classes are in fact "special functions"`

Classes

The most simple class, is an empty class:

```
class Point {}
```

Fields

A bit of a more extensive example is the following:

```
1  class Point {
2   x: number;
3   y: number;
4 }
5
6 const pt = new Point();
7 pt.x = 0;
8 pt.y = 0;
```

A `field` **`declaration`** creates a **public** writeable property on a class.

Constructor

Class constructors are very similar to functions.

You can add **`parameters`** with type **`annotations`**, **`default values`**, and **`overloads`**:

```
class Point {
  x: number;
  y: number;
  // Normal signature with defaults

  constructor(x = 0, y = 0) {
    this.x = x;
    this.y = y;
  }
}
```

Constructor Overloading

Overloading:

```
class Point {
  // Overloads
  constructor(x: number, y: string);
  constructor(s: string);
  constructor(xs: any, y?: any) {
    // TBD
  }
}
```

- Constructors can not have type parameters
  - These belong on the outer class declaration
- Constructors can not have return type annotations

Constructor Overloading

Are you familiar with overloading? Can you explain?

Super Calls

Consider the following **`super`** call:

```
 class Base {
  k = 4;
}

class Derived extends Base {
  constructor() {
    // Prints a wrong value in ES5; throws exception in ES6
    console.log(this.k);
    super();
  }
}
```

Super Calls

Consider the following **`super`** call:

```
1  class Base {
2   k = 4;
3 }
4
5 class Derived extends Base {
6   constructor() {
7     // Prints a wrong value in ES5; throws exception in ES6
8     console.log(this.k);
9     super();
10   }
11 }
```

Important: **`super`** must be called ***`before`*** accessing **`this`** in the constructor of a derived class.

Super Calls

Consider the following **`super`** call:

```
1  class Base {
2    k = 4;
3  }
4
5  class Derived extends Base {
6    constructor() {
7      console.log(this.k);
8      super();
9    }
10 }
```

Prints a wrong value in ES5; throws exception in ES6

Methods

A function property on a class is called a **`method`**. Methods can use all the same **`type annotations`** as functions and constructors:

```
class Point {
  x = 10;
  y = 10;

  scale(n: number): void {
    this.x *= n;
    this.y *= n;
  }
}
```

Getters / Setters

Classes can also have accessors:

```
class C {
    _length = 0;
    get length() {
        return this._length;
    }

    set length(value) {
    this._length = value;
    }
}
```

Implements

You can use an **`implements`** clause to check that a class satisfies a particular interface. An error will be issued if a class fails to implement it correctly:

```typescript
// 05_classes/listings/00_pingeable.ts

interface Pingable {
  ping(): void;
}
class Sonar implements Pingable {
  ping() {
    console.log("ping!");
  }
}
class Ball implements Pingable {
  pong() {
    console.log("pong!");
  }
}
```

What will be the issue with this code?

Extends

Classes may **`extend`** from a base class. A derived class has all the properties and methods of its base class, and also can define additional members.

```
class Animal {
  move() {
    console.log("Moving along!");
  }
}

class Dog extends Animal {
  woof(times: number) {
    for (let i = 0; i < times; i++) {
      console.log("woof!");
    } }
}

const d = new Dog();
// Base class method
d.move();
// Derived class method
d.woof(3);
```

Extends - Exercise

Please take the following code and add a class "Cat". A cat can hiss, but will only do every second time (that is a cat thing ;-)).

Please consider this in your Cat-Method.

The code is prepared to extend in *05_classes/exercises/00_catdog.ts*:

Extends - Exercise

```
class Animal {
  move() {
    console.log("Moving along!");
  }
}

class Dog extends Animal {
  woof(times: number) {
    for (let i = 0; i < times; i++) {
      console.log("woof!");
    } }
}

const d = new Dog();
// Base class method
d.move();
// Derived class method
d.woof(3);
```

Member Visibility

There are three kinds of member visibility available:

- public:
  - the default visibility of class members
  - a public member can be accessed by anything
- protected
  - a protected member is only visible to subclasses of the class it is declared in
- private
  - private is like protected , but does not allow access to the member even from subclasses

Member Visibility

Public:

```
class Greeter {
    public greet() {
        console.log("hi!");
    }
}
const g = new Greeter();
g.greet();
```

Member Visibility

Protected:

```
// 05_classes/listings/01_protected.ts

class Greeter {
  public greet() {
    console.log("Hello, " + this.getName());
  }
  protected getName() {
    return "hi";
  } }
class SpecialGreeter extends Greeter {
  public howdy() {
    // OK to access protected member here
    console.log("Howdy, " + this.getName());
  }
}
const g = new SpecialGreeter();
g.greet(); // OK
g.getName();
```

Question, is g.getName(); ok or will it raise an error?

Member Visibility

It will raise an error:

Property **`getName`** is protected and only accessible **`within class Greeter`** and **`its subclasses`**.

Member Visibility

Private:

```
1  class Base {
2   private x = 0;
3 }
4
5 const b = new Base();
6 // Can't access from outside the class
7 console.log(b.x);
```

Error: Property **`x`** is private and only accessible **`within class Base.`**

Static

Classes may have **`static`** members. These members are not associated with a particular instance of the class.

```
1 class MyClass {
2   static x = 0;
3   static printX() {
4     console.log(MyClass.x);
5   }
6 }
7 console.log(MyClass.x);
8 MyClass.printX();
```

Static members can also use the same **public**, **protected**, and **private** visibility modifiers.

Important: Calling a static member, requires the Class-Name.

Well done!

We can now continue with more advanced topics.

End

That was all for this chapter