

TypeScript OOP Concepts & Modules

Çağatay Sönmez

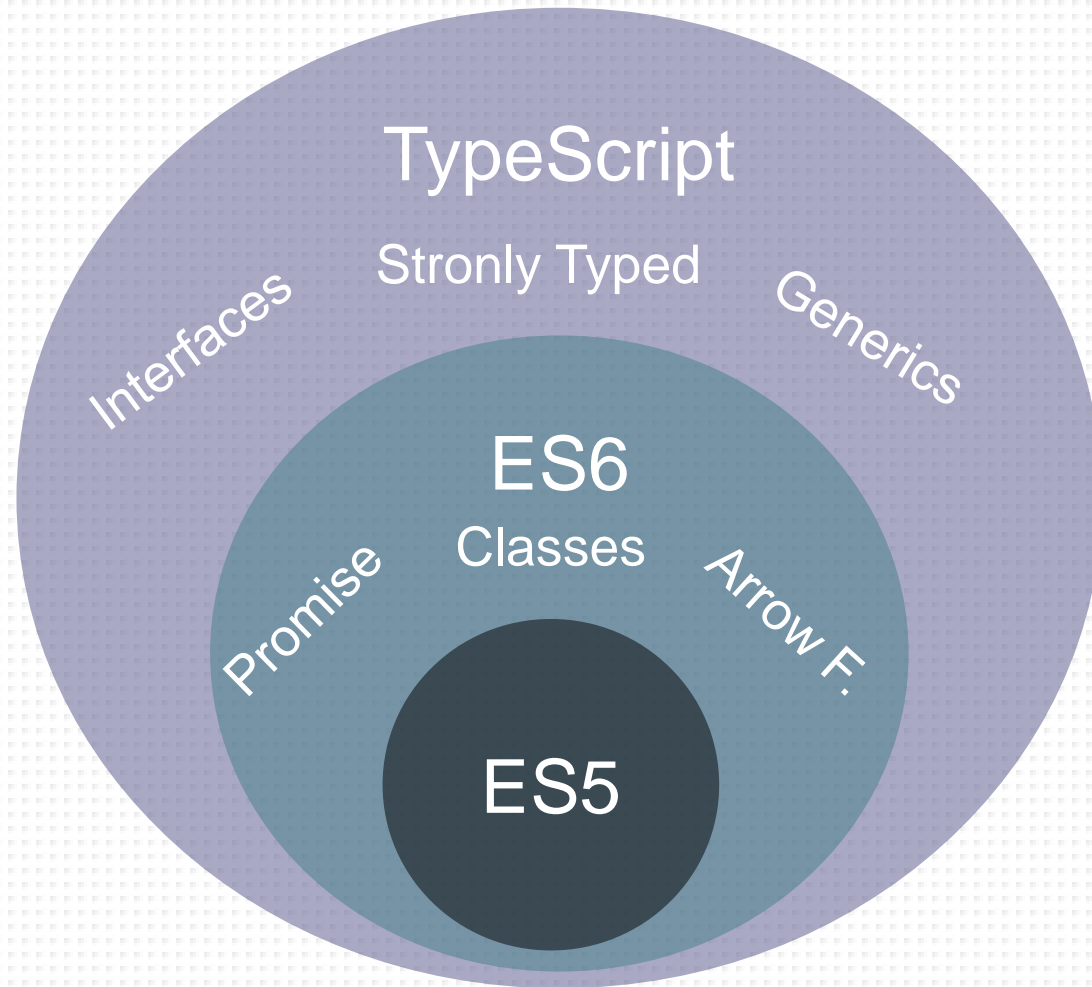
03.10.2021

Agenda

- OOP Concepts of TypeScript
 - Classes
 - Interfaces
 - Inheritance
- Example TypeScript Design Patterns
 - Builder, Adapter, Bridge, Command, Visitor
- TypeScript Modules
 - Internal Modules
 - External Modules

OOP Concepts of TypeScript

A Quick Reminder



TypeScript Class

- TypeScript offers full support for the class keyword introduced in **ES6**

```
1  class Printer {  
2      //field  
3      msg:string;  
4  
5      //constructor  
6      constructor(msg:string) {  
7          this.msg = msg  
8      }  
9  
10     //function  
11     print():void {  
12         console.log(this.msg)  
13     }  
14 }  
15  
16 //create an object and access the function  
17 new Printer("Hello World!").print()
```

Member Visibility

- Private
- Public
 - Default modifier is public
- Protected
 - Only visible to subclasses

```
1  class Printer {  
2      //fields  
3      defaultMsg:string;  
4      private privateMsg: string;  
5      public publicMsg: string = "public message";  
6  
7      //constructor  
8      constructor(msg:string) {  
9          this.privateMsg = msg;  
10         this.defaultMsg = "default message"  
11     }  
12 }  
13  
14 //create an object  
15 var obj = new Printer("Hello World!");  
16  
17 obj.publicMsg = "";  
18 obj.defaultMsg = ""; //default modifier is public!  
19 obj.privateMsg = ""; //Compile Error  
20
```

Static Members

- Static member is not associated with a particular instance of the class

```
1  class Printer {  
2      //fields  
3      public static msg:string;  
4  
5      //function  
6      static print():void {  
7          console.log(this.msg)  
8      }  
9  }  
10  
11  //Accessing static member without class instance  
12  Printer.msg = "Hello World!"  
13  
14  //Invoking static function without class instance  
15  Printer.print();
```

Generic Classes

- Generics let you to create reusable code that work various types

```
1  class Printer<Type> {
2      //fields
3      msg:Type;
4
5      //constructor
6      constructor(msg:Type) {
7          this.msg = msg
8      }
9
10     //function
11     print():void {
12         console.log("Your argument is " + typeof this.msg)
13     }
14 }
15
16 new Printer("").print(); //Prints "Your argument is string"
17 new Printer(12).print(); //Prints "Your argument is number"
```


abstract Classes and Members

- Implementation of abstract classes and functions are not provided as in other object-oriented languages
- The role of abstract classes is to serve as a base class for subclasses

```
1  abstract class Base {  
2      abstract getName(): string;  
3  
4      printName() {  
5          console.log("Hello, " + this.getName());  
6      }  
7  }  
8  
9  const b = new Base(); //Compile Error!  
10                                     //Cannot create an instance of an abstract class.(2511)
```

Inheritance

- TypeScript supports the concept of Inheritance that comes with **ES6**
- TypeScript doesn't support multiple inheritance, like Java

```
1  class Car {  
2      engine:string;  
3  
4      constructor(e:string) {  
5          this.engine = e;  
6      }  
7  }  
8  
9  class BMW extends Car {  
10     getEngine():void {  
11         console.log(this.engine);  
12     }  
13 }  
14  
15 var obj = new BMW("V8");  
16  
17 obj.getEngine(); //Prints V8
```

Interface I

- An interface is a contract which contains only the declaration of the members
- Interfaces are implemented by classes to provide a standard structure

```
1  interface ICar {  
2      engine:string,  
3      setEngine: (e: string) => void  
4  }  
5  
6  class Car implements ICar{  
7      engine:string = "";  
8  
9      setEngine(e: string):void {  
10         this.engine = e;  
11         console.log("engine set to " + e);  
12     }  
13 }  
14  
15 var obj = new Car();  
16 obj.setEngine("V8"); //Prints "engine set to V8"
```

Interface II

- Interfaces are not converted to JavaScript, it's just part of TypeScript

```
1 interface ICar {
2     engine:string,
3     setEngine: (e: string) => void
4 }
5
6 class Car implements ICar{
7     engine:string = "";
8
9     setEngine(e: string):void {
10         this.engine = e;
11         console.log("engine set to " + e);
12     }
13 }
14
15 var obj = new Car();
16 obj.setEngine("V8"); //Prints "engine set to V8"
17
```

```
"use strict";
var Car = /** @class */ (function () {
    function Car() {
        this.engine = "";
    }
    Car.prototype.setEngine = function (e) {
        this.engine = e;
        console.log("engine set to " + e);
    };
    return Car;
})();
var obj = new Car();
obj.setEngine("V8"); //Prints "engine set to V8"
```

Type Casting

- You can use the "as" keyword or <> operator for type castings

```
1  class Car {  
2      printName():void { console.log("Car"); };  
3  }  
4  
5  class BMW extends Car {  
6      printName():void { console.log("BMW"); };  
7  }  
8  
9  var bmw:BMW = new BMW();  
10  
11 var car1:Car = new Car();  
12 car1.printName();           //Prints "Car"  
13  
14 var car2:Car = bmw as Car;   //same as <Car>bmw  
15 car2.printName();           //Prints "BMW"
```

Why TypeScript 😊

```
1 class Car {  
2     printName():void {  
3         console.log("Car");  
4     };  
5 }  
6  
7 class BMW extends Car {  
8     printName():void {  
9         console.log("BMW");  
10    };  
11 }  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41
```

```
var __extends = (this && this.__extends) || (function () {  
    var extendStatics = function (d, b) {  
        extendStatics = Object.setPrototypeOf ||  
            ({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||  
            function (d, b) { for (var p in b) if (Object.prototype.hasOwnProperty.call(b, p)) d[p] = b[p]; };  
        return extendStatics(d, b);  
    };  
    return function (d, b) {  
        if (typeof b !== "function" && b !== null)  
            throw new TypeError("Class extends value " + String(b) + " is not a constructor or null");  
        extendStatics(d, b);  
        function __() { this.constructor = d; }  
        d.prototype = b === null ? Object.create(b) : ().__proto__ = b.prototype, new __();  
    };  
})();  
var Car = /** @class */ (function () {  
    function Car() {  
    }  
    Car.prototype.printName = function () {  
        console.log("Car");  
    };  
    ;  
    return Car;  
})();  
var BMW = /** @class */ (function (_super) {  
    __extends(BMW, _super);  
    function BMW() {  
        return _super !== null && _super.apply(this, arguments) || this;  
    }  
    BMW.prototype.printName = function () {  
        console.log("BMW");  
    };  
    ;  
    return BMW;  
})(Car);
```

OOP Concepts in a Nutshell

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

```
1  abstract class Animal{
2    |  abstract getName(): string;
3  }
4
5  class Dog extends Animal{
6    |  private hasTail: boolean = true;
7
8    |  getName(): string { return "Dog"; }
9
10   |  getTail(): boolean { return this.hasTail;}
11  }
12
13  class Cat extends Animal{
14    |  getName(): string { return "Cat"; }
15  }
16
17  var cat: Cat = new Cat();
18  var dog: Dog = new Dog();
19
20  function printName(animal: Animal){
21    |  console.log(animal.getName());
22  }
23
24  printName(cat);    //Prints "Cat"
25  printName(dog);    //Prints "Dog"
```

OOP Concepts in a Nutshell

➤ Abstraction

```
1 abstract class Animal{  
2   | abstract getName(): string;  
3   }  
4
```

➤ Encapsulation

```
5 class Dog extends Animal{  
6   | private hasTail: boolean = true;  
7  
8   getName(): string { return "Dog"; }  
9  
10  getTail(): boolean { return this.hasTail;}  
11  }  
12
```

➤ Inheritance

```
13 class Cat extends Animal{  
14   | getName(): string { return "Cat"; }  
15   }  
16
```

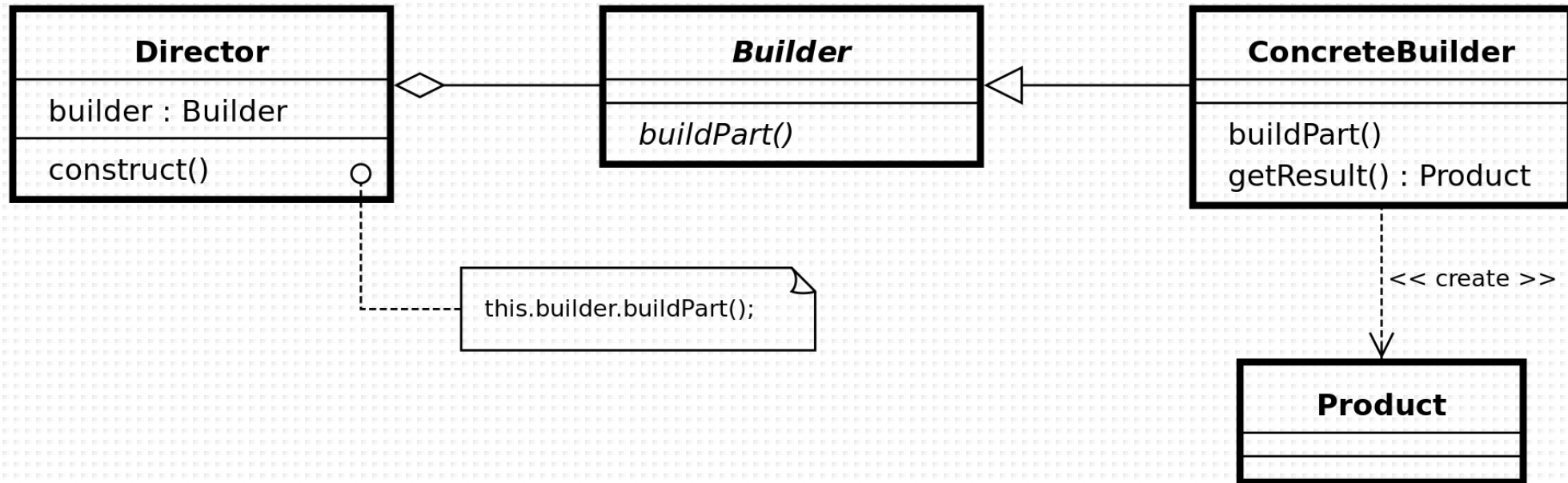
➤ Polymorphism

```
17 var cat: Cat = new Cat();  
18 var dog: Dog = new Dog();  
19  
20 function printName(animal: Animal){  
21   | console.log(animal.getName());  
22   }  
23  
24 printName(cat);    //Prints "Cat"  
25 printName(dog);    //Prints "Dog"
```


Example TypeScript Design Patterns

Builder Pattern / Creational

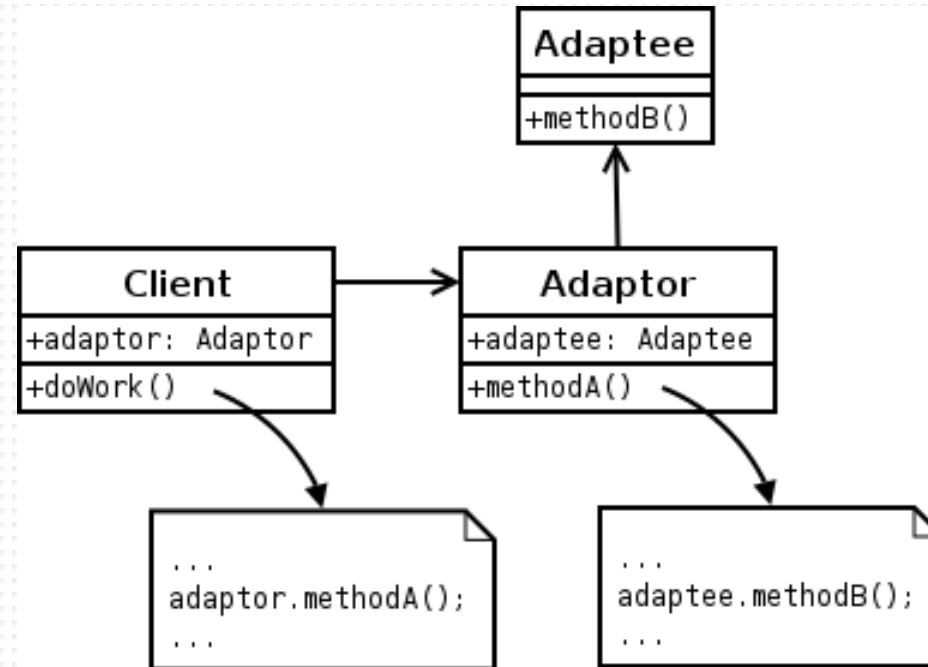
- Click [here](#) to see the source code on GitHub



source: https://en.wikipedia.org/wiki/Builder_pattern

Adapter Pattern / Structural

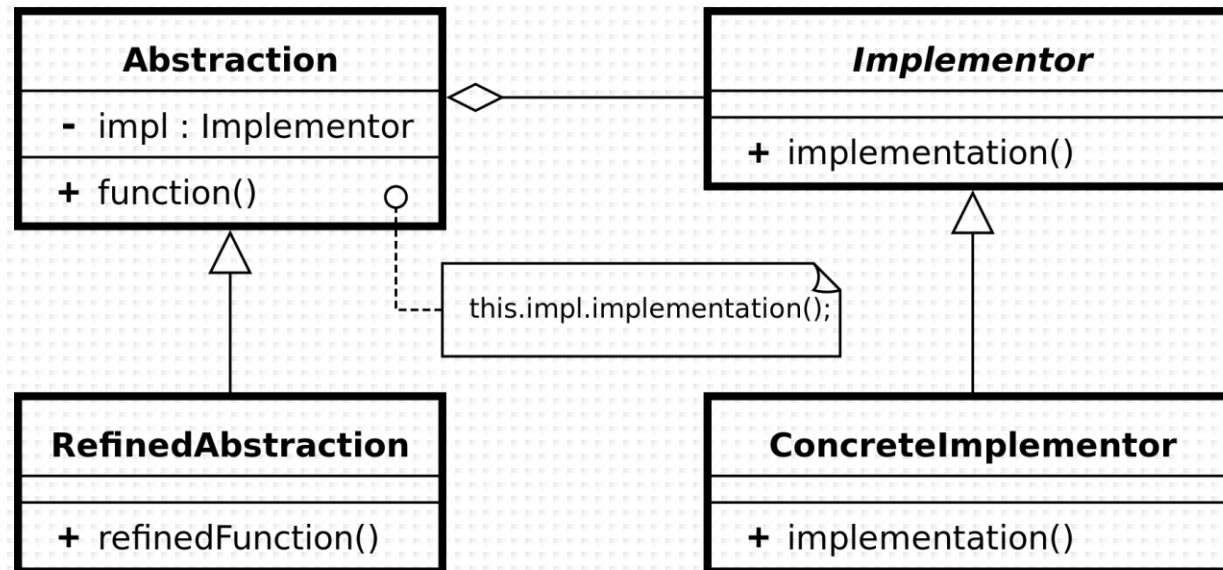
- Click [here](#) to see the source code on GitHub



source: https://en.wikipedia.org/wiki/Adapter_pattern

Bridge Pattern / Structural

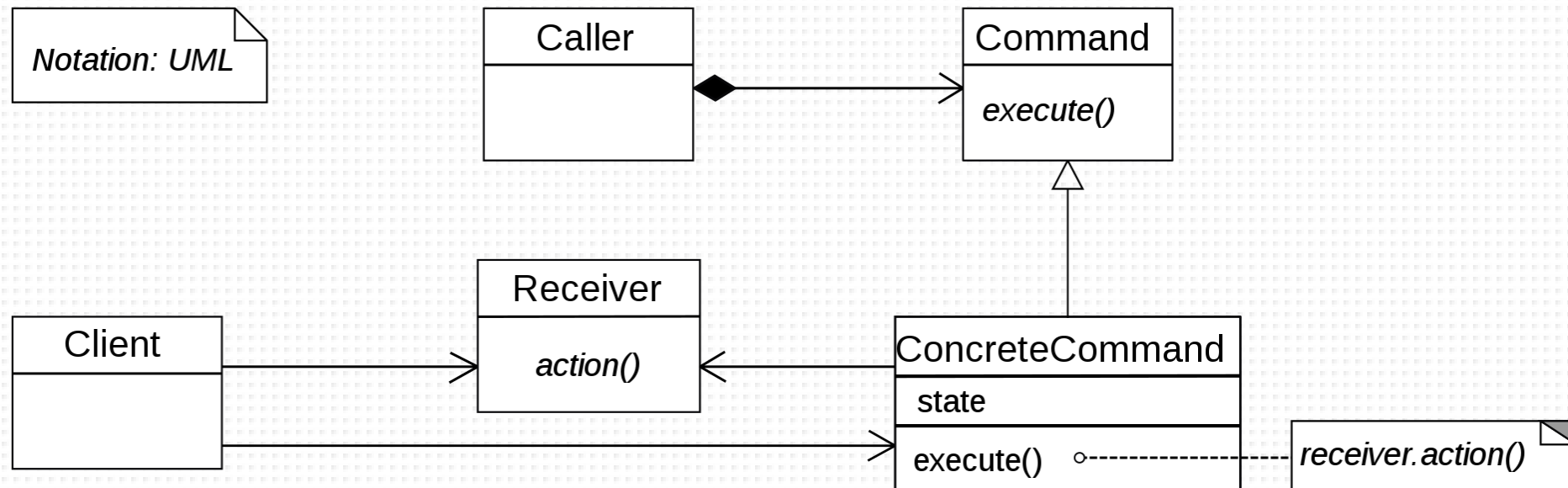
- Click [here](#) to see the source code on GitHub



source: https://en.wikipedia.org/wiki/Bridge_pattern

Command Pattern / Behavioral

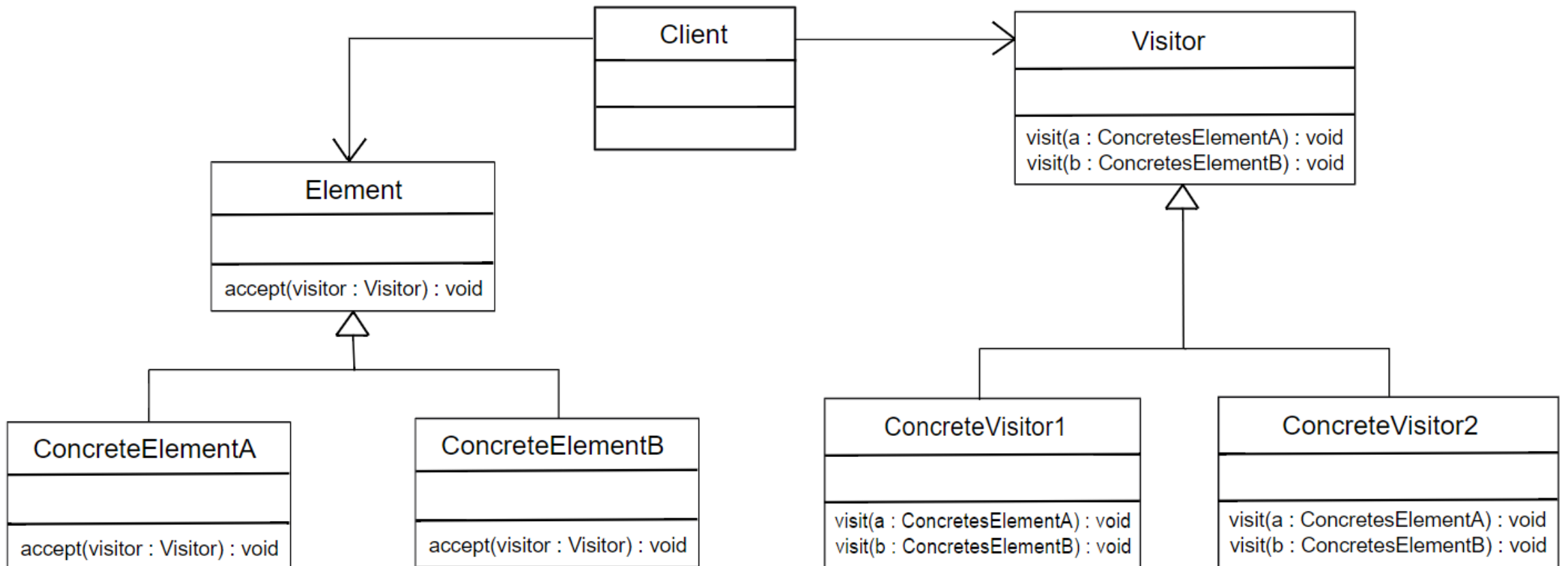
- Click [here](#) to see the source code on GitHub



source: https://en.wikipedia.org/wiki/Command_pattern

Visitor Pattern / Behavioral

- Click [here](#) to see the source code on GitHub



TypeScript Modules

Modules

- Modules creates a group of related variables, functions, classes, and interfaces
- Modules provides many advantages in big projects
 - Separation of concerns
 - Reusability
 - Maintainability
 - Testability



Spaghetti Code

VS



Ravioli Code

Modules Types

- Modules are declared with «module» keyword, and the functions in the modules are exported with «export» keyword
- There are 2 modules type
 - Internal Modules
 - External modules

```
1  ✓ function add(a: number, b: number) {  
2    |   console.log("Sum: " +(a+b));  
3    |  
4    | }  
5  
6    add(5, 10);  
7  
8
```

VS

```
1  ✓ module Utils {  
2    ✓ export function add(a: number, b: number) {  
3      |   console.log("Sum: " +(a+b));  
4      | }  
5    }  
6  
7    Utils.add(5, 10);  
8
```

Global module (window)

Utils module

Global Scope vs Module Scope

```
1 class Math {
```

⊗ input.ts 1 of 2 problems

Duplicate identifier 'Math'. (2300)

lib.es5.d.ts(617, 11): 'Math' was also declared here.
lib.es5.d.ts(726, 13): and here.

```
2     constructor() {  
3         console.log("Math class created!");  
4     }  
5 }  
6  
7 var math = new Math();  
8
```

VS

```
1  
2  
3  
4 namespace Utils {  
5     export class Math {  
6         constructor() {  
7             console.log("Math class created!");  
8         }  
9     }  
10 }  
11  
12 var math = new Utils.Math();  
13  
14  
15
```

Internal Modules

- Internal modules are used for gruppung code and **not needed** to be imported
- After TypeScript 1.5 internal modules are replaced with **namespaces**
- You can still use module keyword, but it is **not** recommended

```
1 namespace Utils {  
2   export class Math {  
3     public sum(a: number, b: number) {  
4       console.log("Sum: " +(a+b));  
5     }  
6   }  
7 }  
8  
9 let math = new Utils.Math();  
10 math.sum(5, 10);  
11  
12  
13  
14  
15  
16  
17
```

Immediately Invoked
Function Expression (IIFE)

```
var Utils;  
(function (Utils) {  
  var Math = /** @class */ (function () {  
    function Math() {  
    }  
    Math.prototype.sum = function (a, b) {  
      console.log("Sum: " + (a + b));  
    };  
    return Math;  
  })();  
  Utils.Math = Math;  
})(Utils || (Utils = {}));  
var math = new Utils.Math();  
math.sum(5, 10);
```

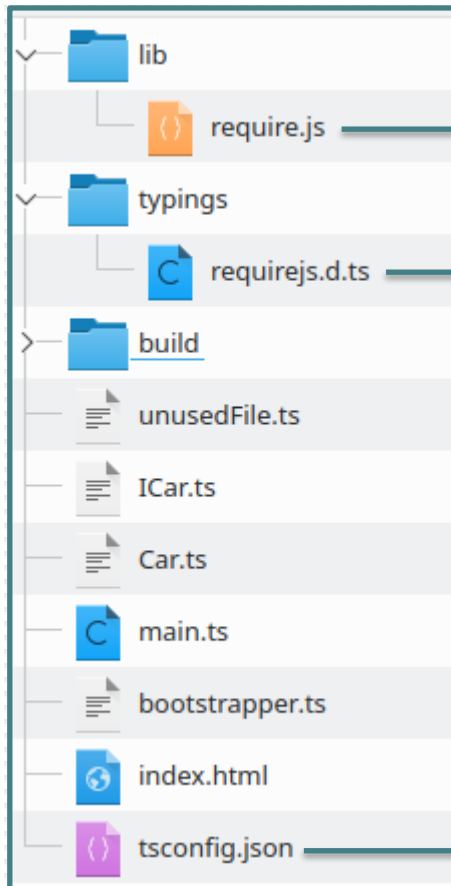
External Modules

- External modules provides modularity among different files
- External modules **do not** use the **module** keyword
- A module can be created using the **export** keyword
- A module can be used in another module using the **import** keyword
- You should use correct module loader based on your project, e.g.
 - CommonJS option for server side Node.js applications
 - AMD option for client-side module loader require.js

External Modules with RequireJS

- CommonJS is generally used server-side, so it doesn't fit in the browser environment very well
- AMD (Asynchronous Module Definition) is generally more used in client-side (in-browser)
- The main difference between CommonJS and AMD is asynchronous loading of module dependencies.
- RequireJS is one of the most popular AMD implementations

External Modules with RequireJS - Setup



1- download RequireJS library

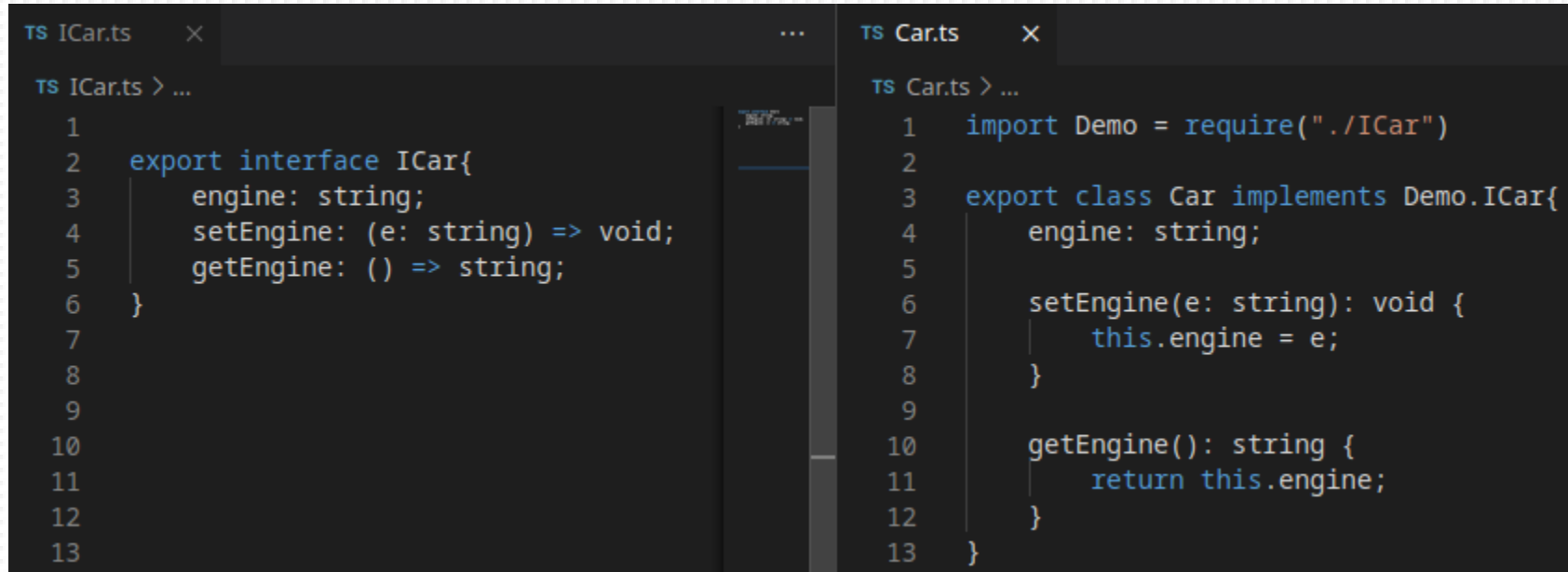
2- download RequireJS d.ts file

3- enable AMD module loader

```
tsconfig.json x
tsconfig.json > ...
1 {
2   "compilerOptions": {
3     "target": "es5",
4     "module": "AMD",
5     "sourceMap": true,
6     "outDir": "build"
7   }
8 }
```

External Modules with RequireJS - Develop

- Export/import your modules



```
TS ICar.ts  x
TS ICar.ts > ...
1
2 export interface ICar{
3     engine: string;
4     setEngine: (e: string) => void;
5     getEngine: () => string;
6 }
7
8
9
10
11
12
13

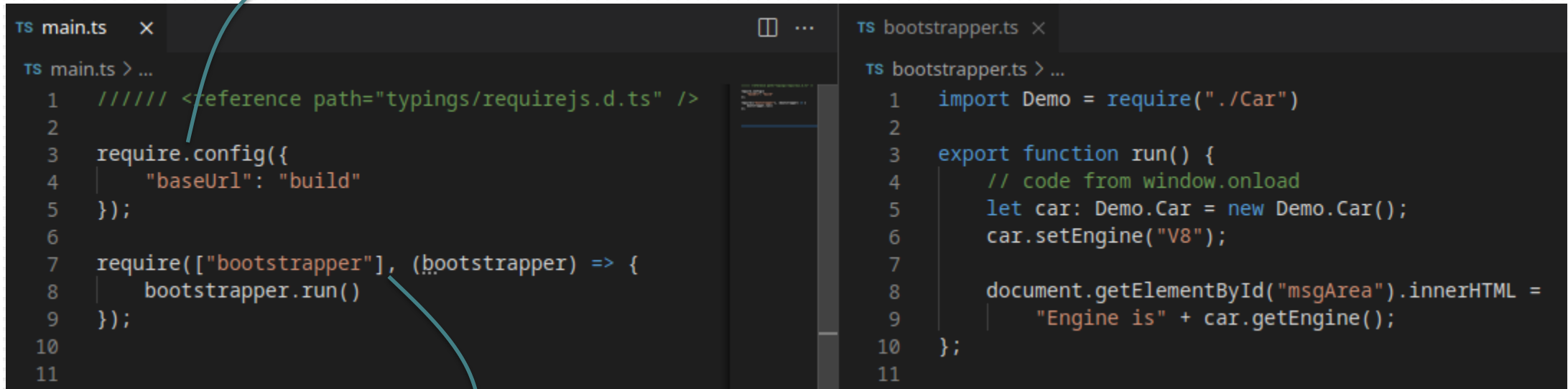
TS Car.ts  x
TS Car.ts > ...
1 import Demo = require("../ICar")
2
3 export class Car implements Demo.ICar{
4     engine: string;
5
6     setEngine(e: string): void {
7         this.engine = e;
8     }
9
10    getEngine(): string {
11        return this.engine;
12    }
13 }
```

Exported Interface

Exported Class

External Modules with RequireJS - Configure

1- configure RequireJS



```
TS main.ts x
TS main.ts > ...
1  //////// <reference path="typings/requirejs.d.ts" />
2
3  require.config({
4    "baseUrl": "build"
5  });
6
7  require(["bootstrapper"], (bootstrapper) => {
8    bootstrapper.run()
9  });
10
11

TS bootstrapper.ts x
TS bootstrapper.ts > ...
1  import Demo = require("../Car")
2
3  export function run() {
4    // code from window.onload
5    let car: Demo.Car = new Demo.Car();
6    car.setEngine("V8");
7
8    document.getElementById("msgArea").innerHTML =
9      "Engine is" + car.getEngine();
10 };
11
```

2- load bootstrapper and its dependencies

3- run your main function

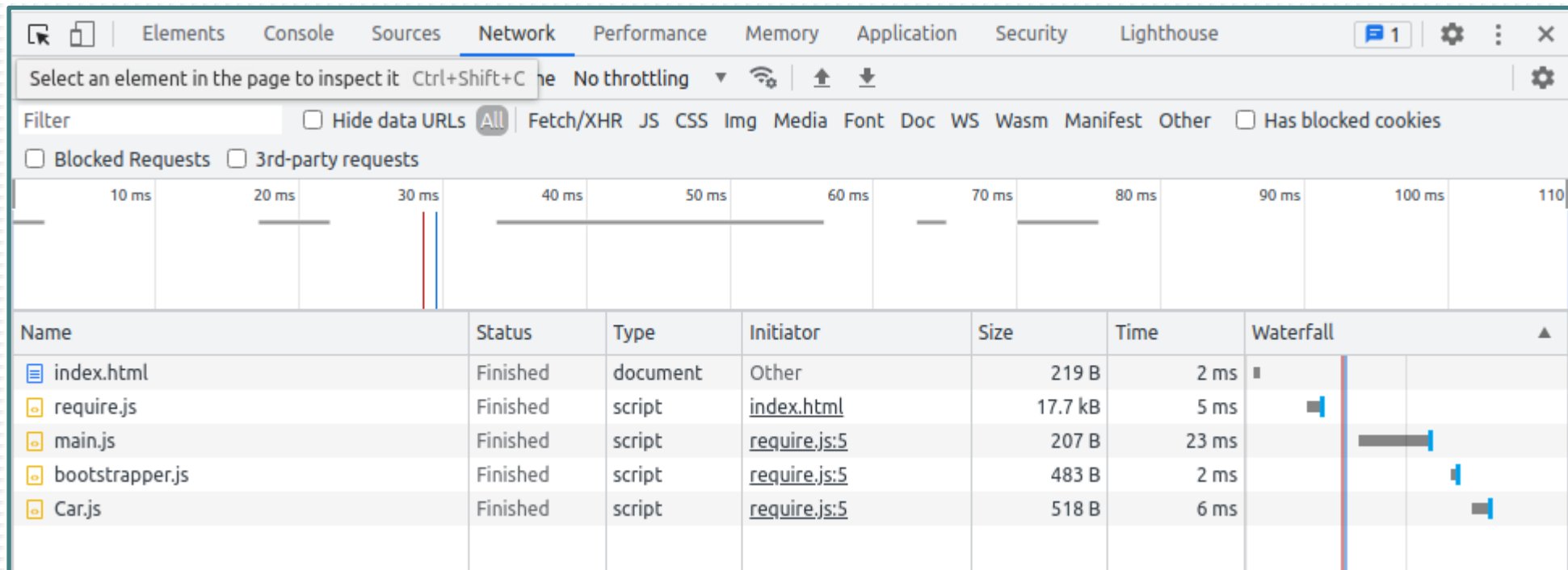
External Modules with RequireJS - Execute

- Load only request.js in the HTML file, let other JS files to be loaded dynamically
- *data-main* attribute tells request.js to load build/main.js after request.js is loaded

```
<> index.html X
<> index.html > ...
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Hello World Example</title>
5      <script data-main="build/main" src="lib/require.js"></script>
6    </head>
7    <body>
8      <p id="msgArea"></p>
9    </body>
10 </html>
11
12
```

External Modules with RequireJS - Load

- The required JavaScript files are loaded dynamically when they are used
- Unused JavaScript files are not loaded.



QUESTIONS?