Modules & TypeScript

CS445 Modern Asynchronous Programming

Maharishi University of Management
Department of Computer Science

Teaching Faculty: Assistant Professor Umur Inan

Prepared by: Associate Professor Asaad Saad

ES6 Modules

Within any JS module, everything is considered private, until we export it, we can have two kinds of exports:

- `export default` (can be used once) *default export*
- `export` (can be used multiple times) *named export*

To import what is explicitly exported we use:

```
import varForDefault, {desctructuredExports} from './module.js'
```

ES6 Modules in the Browser

To use a JS file in the browser that imports JS modules, we must add type attribute to indicate that this is JS file is using modules:

```
<script src="app.js" type="module"></script>
```

Only modern browsers support modules. As a fallback, we can still bundle the code and provide one JS file for the application.

Modules in NodeJS

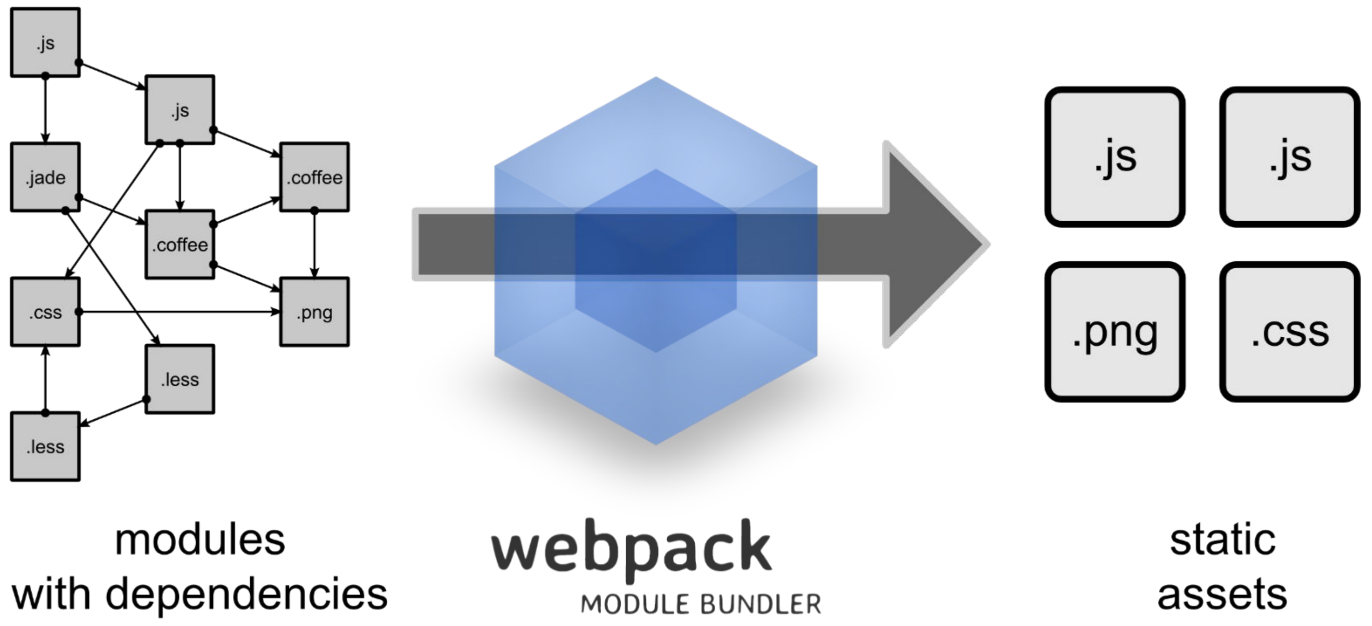
Node uses common JS specifications as follows:

- `module.exports`
- `const obj = require('./module.js');`

ES6 modules are supported in Node, but they still experimental, they have `.mjs` extension.

Webpack – Module Bundler

It bundles our code into one JS file. This file will still need compilation in order to make our app to work (to be done later in the browser JiT).



Using Webpack Bundler

Within your project folder, run the following commands to install webpack as a development dependency:

```
npm init -y
```

You need Node to be installed

```
npm i webpack webpack-cli –D (or --save-dev)
```

Add the following command to scripts in **`package.json`**:

```
"scripts": {
    "build": "webpack ./app.js -o ./dist -w",
 },
```

Run the following command to bundle:

```
npm run build
```

TypeScript

TypeScript is an open-source object-oriented language developed and maintained by **Microsoft**. It is a typed superset of JavaScript that compiles to plain JavaScript.

TypeScript was first released in October 2012.

It's the official language adopted by the Google Angular Team to write Angular projects.

Official website: https://www.typescriptlang.org

Source code: https://github.com/Microsoft/TypeScript

Why TypeScript

JavaScript is a dynamic programming language with **no type** system.

A no type system means that a variable in JavaScript can have any type of value such as string, number, boolean etc.

The type system increases the code quality, readability and makes it easy to maintain and refactor code base.

Monomorphic code can be optimized by the JS Engine.

Errors can be caught at compile time (development) rather than at run time.

Enterprise Web Applications

Without the type system, it is difficult to scale JavaScript to build complex applications with large teams working on the same code.

The reason to use TypeScript is that it allows JavaScript to be used at scale.

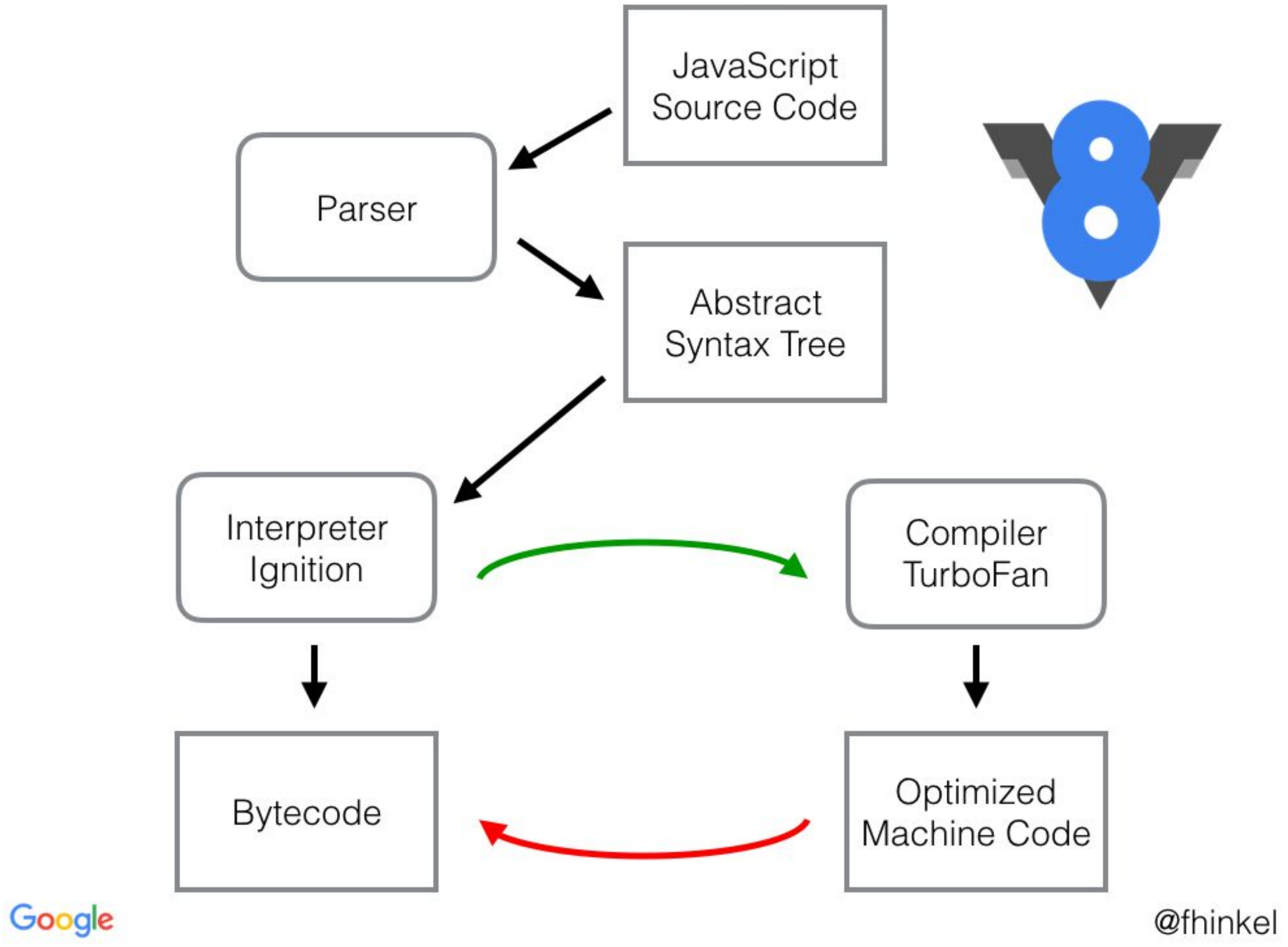
Why Types?

One of the great things about type checking is that:

1. It helps writing safe code because it can prevent bugs at compile time.
2. Compilers can improve and run the code faster.

It's worth noting that types are **optional** in TypeScript.

V8


JavaScript Source Code
Parser
Abstract Syntax Tree
Interpreter Ignition
Compiler TurboFan
Bytecode
Optimized Machine Code
Google
@fhinkel

TypeScript Compiler

TypeScript compiles into simple JavaScript.

A TypeScript code is written in a file with `.ts` extension and then compiled into JavaScript using the TypeScript compiler.

A TypeScript compiler needs to be installed on your platform. Once installed, the command **`tsc filename.ts`** compiles the TypeScript code into a plain JavaScript file.

TypeScript Features

- Cross-Platform
- Object Oriented Language
- Static type-checking
- Optional Static Typing
- DOM Manipulation
- ES6/next Features

Setup

```
npm install -g typescript
tsc -v
tsc filename.ts -w
```

ⓘ You need Node to be installed

Microsoft provides Visual Studio Code with TypeScript support built in.

https://www.typescriptlang.org/play

tsconfig.json

To create a TypeScript configuration file:

```
tsc --init //it creates a tsconfig.json
```

Once you create a config file, we could simply type **`tsc`** and it's going to automatically find all **`*.ts`** and compile them to JavaScript.

```
{
    "compilerOptions": {
     "module": "commonjs",
     "target": "es5",
     "noImplicitAny": false,
          "noEmitOnError": true,
     "sourceMap": true,
          "outDir": "./js",
    },
    "exclude": [ "node_modules" ]
}
```

tsconfig.json

Click here for full list of options

First Example

```
function add(a: number, b: number) {
    return a + b;
}
const sum1: number = add(10,15);

const sum2: number = add('WAD2', 5);
```

Transpile the code to JS.

Type Annotations

We can specify the type using **`:type`** after the name of the variable, parameter or property.

TypeScript includes all the primitive types of JavaScript- number, string and boolean.

```
const grade: number = 90; // number variable
const name: string = "Asaad"; // string variable
const isFun: boolean = true; // Boolean variable
```

Type Annotation of Parameters

```
function hello(id:number, name:string) {
    console.log(`Id = ${id}, Name = ${name}`);
}
```

Type annotations are used to enforce type checking. It is not mandatory in TypeScript to use type annotations. Type annotations help the compiler in checking types and helps avoid errors dealing with data types.

Type Annotation in Object

```
let employee : {
    id: number;
    name: string;
};

employee = {
  id: 100,
  name : "John"
}
```

If you try to assign a string value to `id` then the TypeScript compiler will give the following error:

```
error TS2322: Type '{ id: string; name: string; }' is not assignable to type '{ id:number; name: string; }'.
Types of property 'id' are incompatible. Type 'string' is not assignable to type 'number'.
```

number, string, and boolean

```
let grade:number = 90; // number
let n = new Number(90);
console.log(n) // Output: a number object with value 90
console.log(typeof n) // Output: object

let n2 = n.valueOf(); // returns the primitive value of the number
console.log(n2) //Output: 90
console.log(typeof n2) //Output: number

let employeeName:string = 'Asaad Saad';
let isPresent:boolean = true;
```

Note that, **Number** with an uppercase N is different from **number** with a lowercase n. Upper case **Number** is an object type whereas lower case **number** is a primitive type.

Union Type

Union type allows us to use more than one data type for a variable or a function parameter.

```
(type1 | type2 | type3 | .. | typeN)
```

```
let course: (string | number);
let data: string | number;
function process(code: (string | number)){}
```

Array

There are two ways to declare an array:

1. Using **square brackets**

```
let values: number[] = [12, 24, 48];
```

2. Using a **generic array type**, `Array<elementType>`

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

You can always initialize an array with many data types elements, but you will not get the advantage of TypeScript's type system.

Multi Type Array

An array in TypeScript can contain elements of different data types.

```
let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
// or
let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
```

Tuple

Tuple is a new data type where a variable can include multiple data types in the specified array position.

```
let user: [number, string, boolean, number, string];
user = [1, "Asaad", true, 20, "Faculty"];

let family: [number, string][];
family = [[1, "Asaad"], [2, "Mike"], [3, "Mada"]];
```

Enum

Enums allow us to declare **a set of named Constants**, a collection of related values that can be numeric or string values.

Enum values start from zero and increment by 1 for each member.

Enum in TypeScript supports **reverse mapping**.

```
enum Technologies {
  Angular,
  React,
  ReactNative
}

// Technologies.React; returns 1
// Technologies["React"]; returns 1
// Technologies[0]; returns Angular
```

```
console.log(Technologies);

{
    '0': 'Angular',
    '1': 'React',
    '2', 'ReactNative',
    Angular: 0,
    React: 1,
    ReactNative: 2
}
```

Any

When you do not have prior knowledge about the type of some variables and deal with dynamic content, we can **any** type.

```
let something: any = 'Asaad';
something = 569;
something = true;

let data: any[] = ["Asaad", 569, true];
```

Avoid using **any** in your code as you lose the type.

Type Inference

It is not mandatory to annotate types in TypeScript, as it infers types of variables when there is no explicit information available in the form of type annotations.

```
let text = "some text";
text = 123; // Type '123' is not assignable to type 'string'
```

Type inference in complex objects

TypeScript looks for the most common type to infer the type of the object.

```
let data = [1, 2, "Asaad"];

data.push(3);
data.push(true); // Type 'true' is not assignable to type 'string | number'
```

Type Assertion

Type assertion allows you to set the type of a value and tells the compiler **not to infer** it. *(similar to type casting)*

```
let code: any = 123;
let courseCode = <number> code;

console.log(typeof(courseCode)); // number
```

Type Assertion with Object

```
// the compiler assumes that the type of employee is {} with no properties.
let employee = {};
employee.name = 'Asaad'; // Compiler Error: Property 'name' does not exist on type '{}'
```

Interfaces are used to define the structure of variables. TS compiler will autocomplete employee properties

```
type Employee = { name: string } // We can also create an Interface

let employee = <Employee>{};
employee.name = 'Asaad'; // OK
```

There are two ways to do Type Assertion

1. Using the angular bracket **<>** syntax

```
let code: any = 123;
let courseCode = <number> code;
```

2. Using **as** keyword

```
let code: any = 123;
let courseCode = code as number;
```

Function Parameters

In TypeScript, the compiler expects a function to receive the exact number and type of arguments as defined in the function signature.

The parameters that may or may not receive a value can be appended with a '**?**' to mark them as optional.

```
function Sum(x: number, y: number) :number {
    return x + y;
}

function Greet(greeting: string, name?: string  = "my dear" ) :string {
    return greeting + ' ' + name + '!';
}

// function type
let myFunction: (a: number, b: number) => number;
```

Interface

Interface is a structure that defines the **contract** in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

An interface is defined with the keyword interface and it can include properties and method declarations using a function or an arrow function.

```
interface IEmployee {
    empCode: number;
    empName: string;
    getSalary: (number) => number;
    getManagerName(number): string;
}
```

Interface as Type

Interface in TypeScript can be used to **define a type** and also to **implement** it in the class. We can have optional properties, marked with a "?". We can mark a property as read only.

```
interface IKeyPair {
    readonly key: number;
    value?: string;
}

let kv1: IKeyPair = { key:1, value:"Asaad" };
let kv2: IKeyPair = { key:2 };
kv2.key = 3; // Compiler error
```

Extending Interfaces

Interfaces can extend one or more interfaces. The object from the extended interface **must include all the properties and methods from both interfaces**, otherwise, the compiler will show an error.

```
interface ICity {
    name: string;
}

interface IZipcode extends ICity {
    zipcode: number;
}

let northStreet: IZipcode = {
    zipcode:52557,
    name:"Fairfield",
}
```

Implementing an Interface

Interfaces can be implemented with a Class. The Class implementing the interface needs to **strictly conform to the structure of the interface**.

The implementing class can define extra properties and methods, but at least it must define all the members of an interface.

Implements Example

```
interface ICourse {
    code: number;
    name: string;
    getGrade:(number)=>number;
}

class Course implements ICourse {
    code: number;
    name: string;
    constructor(code: number, name: string) {
                this.code = code;
                this.name = name;
    }
    getGrade(code:number):number {
        return 90;
    }
}

let course = new Course(445, "Modern Asynchronous Programming");
```

Class

Classes are the fundamental entities used to create reusable objects. Functionalities are passed down to other classes and objects can be created from classes.

The class in TypeScript is compiled to plain JavaScript function constructor by the TS compiler to work across platforms and browsers.

A class can include the following:

- Constructor → speciad Beha
- Properties — Data
- Methods — Behavior

Constructor

The constructor is a special method which is called when creating an object. An object of the class can be created using the **new** keyword.

It is not necessary for a class to have a constructor.

Inheritance

TypeScript classes can be extended to create new classes with inheritance, using the **extends** keyword.

sub sup

```
class B extends A {}
```

This means that the B class now includes all the members of the A class.

The constructor of the B class initializes its own members as well as the parent class's properties using the **super** keyword.

Inheritance Example

```
class Course {
    name: string;
    constructor(name: string) { this.name = name }
}

class MSD extends Course {
    code: number;
    constructor(code: number, name:string) {
        super(name);
        this.code = code;
    }
    displayName():void {
        console.log(`Name = ${this.name}, Course Code = CS${this.code}`);
    }
}

let course = new MSD(445, "Modern Asynchronous Programming");
course.displayName(); // Name = Modern Asynchronous Programming, Course Code = CS445
```

We must call **super()** method first before assigning values to properties in the constructor of the derived class.

A class can implement multiple interfaces

```
class MSD implements ICourse, ICode {
    code: number;
    name: string;

    constructor(code: number, name:string) {
        this.code = code;
        this.name = name;
    }

    display(): void {
        console.log(`${this.name}, Course Code = CS${this.code}`);
    }
}

let wad: MSD = new MSD(445, "Modern Asynchronous Programming");
wad.display(); // Modern Asynchronous Programming, Course Code = CS445
```

The **MSD** class implements two interfaces - **`ICourse`** and **`ICode`**. So, an instance of the **MSD** class can be assigned to a variable of **`ICourse`** or **`ICode`** type. However, an object of type **`ICode`** cannot call the **`display()`** method because **`ICode`** does not include it.

```
interface ICourse {
    name: string;
    display():void;
}

interface ICode {
    code: number;
}
```

Method Overriding

```
class Meditator {
    name: string;
    constructor(name: string) {this.name = name }
    meditate(duration:number = 20) {
        console.log(this.name + " is meditating for " + duration + " mins!");
    }
}

class Sidha extends Meditator {
    constructor(name: string) {super(name)}
    meditate(duration:number = 40) {
        console.log('Meditation started')
        super.meditate(duration);
    }
}

let asaad = new Sidha("Asaad");
asaad.meditate();  // Meditation started Asaad is meditating for 40 mins!
```

When a child class defines its own implementation of a method from the parent class, it is called method overriding.

Abstract Class

Abstract classes are mainly for inheritance where other classes may derive from them. **We cannot create an instance of an abstract class**.

An abstract class includes one or more **`abstract` methods or properties**.

The class which extends the abstract class **must** define all the abstract methods.

Mostly used when child classes want to share the some but not all behavior, it should be used primarily for objects that are closely related.

Abstract Class Example

```
abstract class Course {
    faculty: string;
    abstract name: string;
    constructor(faculty: string) { this.faculty = faculty }
    abstract checkPrerequisite(string): boolean;
}

class MAP extends Course {
    name: string;
    code: number;
    constructor(faculty: string, name: string, code: number) {
        super(faculty); // must call super()
        this.name = name;
    this.code = code;
    }
    checkPrerequisite(faculty: string): boolean {
        // custom check
    }
}
```

The class which implements an abstract class must call super() in the constructor.

Access Modifiers

There are three types of access modifiers: **`public`**, **`private`** and **`protected`**. Encapsulation is used to control class members' visibility.

`public`

By default, all members of a class in TypeScript are public. All the public members can be accessed anywhere without any restrictions.

```
class Course {
    public code: string;
    name: string;
}

let course = new Course();
course.code = "CS445";
course.name = "MAP";
```

code and **name** are accessible outside of the class using an object of the class.

private

The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

```
class Course {
    private code: string;
    name: string;
}

let course = new Course();
course.code = "CS445"; // Compiler Error
course.name = "MAP"; // OK
```

`protected`

The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.

Protected Example

Property **code** is protected and only accessible within class **Course** and its subclasses.

```
class Course {
    public name: string;
    protected code: number;
    constructor(name: string, code: number){
        this.name = name;
        this.code = code;
    }
}

class MAPCourse extends Course{
    private details: string;
    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.details = `${department} - ${this.code}`;
    }
}

let map = new MAPCourse("Modern Asynchronous Programming", 445, "Computer Science");
map.code; // Compiler Error
```

Readonly

Read-only members can be accessed outside the class, but their value cannot be changed. Since read-only members cannot be changed outside the class, they either need to be initialized at declaration or initialized inside the class constructor.

```
class Course {
    readonly code: number;
    name: string;
    constructor(code: number, name: string){
        this.code = code;
        this.name = name;
    }
}
let course = new Course(569, "WAD");
course.code = 445; // Compiler Error
course.name = 'Modern Asynchronous Programming'; // Ok
```

Static

ES6 includes static members and so does TypeScript. The static members of a class are accessed using the class name and dot notation, without creating an object.

```
class Circle {
    static pi: number = 3.14;

    static calculateArea(radius:number) {
        return this.pi * radius * radius;
    }
}
Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5
```

Example: Look at the **Date** function constructor in JavaScript.

Class Example

```
interface Book {
    bookName: string;
    isbn: number;
}
class Course {
    name: string;
    code: number;

    constructor(name: string, code: number) {
        this.name = name;
        this.code = code;
    }

    useBook(book: Book) {
        console.log(`Course ${this.name} is using the textbook:
                 ${book.bookName} who's ISBN = ${book.isbn}`);
    }
}
```

Constructor methods must be named **`constructor`**. They can optionally take parameters but they can’t return any values. When a class has no constructor defined explicitly, one will be created automatically.

Class Example - Shortcut

Adding **access modifiers** to the constructor arguments lets the class know that they're properties of a class. If the arguments don't have access modifiers, they'll be treated as an argument for the constructor function and not properties of the class.

```
interface Book {
    bookName: string;
    isbn: number;
}
class Course {

    constructor(public name: string, public code: number) {}

    useBook(book: Book) {
        console.log(`Course ${this.name} is using the textbook:
                 ${book.bookName} who's ISBN = ${book.isbn}`);
    }
}
```