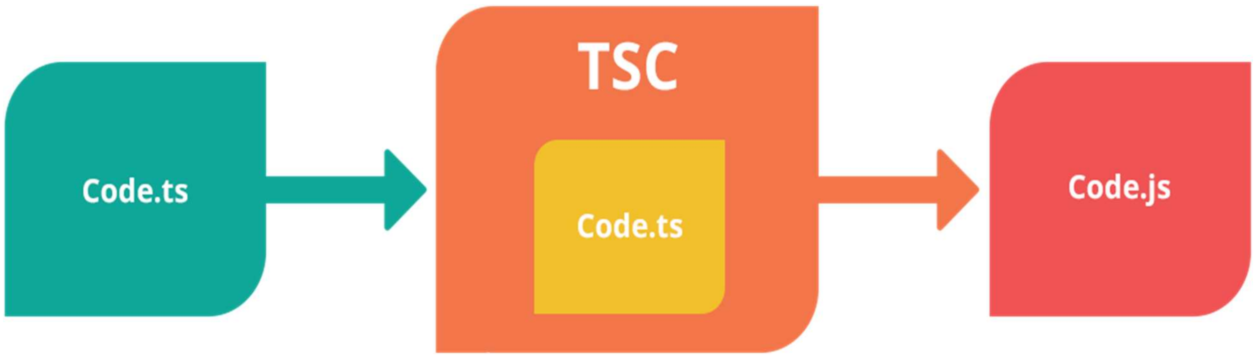
WEB DEVELOPMENT USING MEAN STACK

Unit 3 - Basics of Angular js

Typescript?

- TypeScript is an object-oriented programming language developed and maintained by Microsoft Corporation. It is a superset of JavaScript and contains all of its elements and we can say that TypeScript is modern JavaScript with classes, optional types, interfaces, and more.

- TypeScript totally follows the OOPS concept and with the help of TSC (TypeScript Compiler), we can convert Typescript code (.ts file) to JavaScript (.js file)

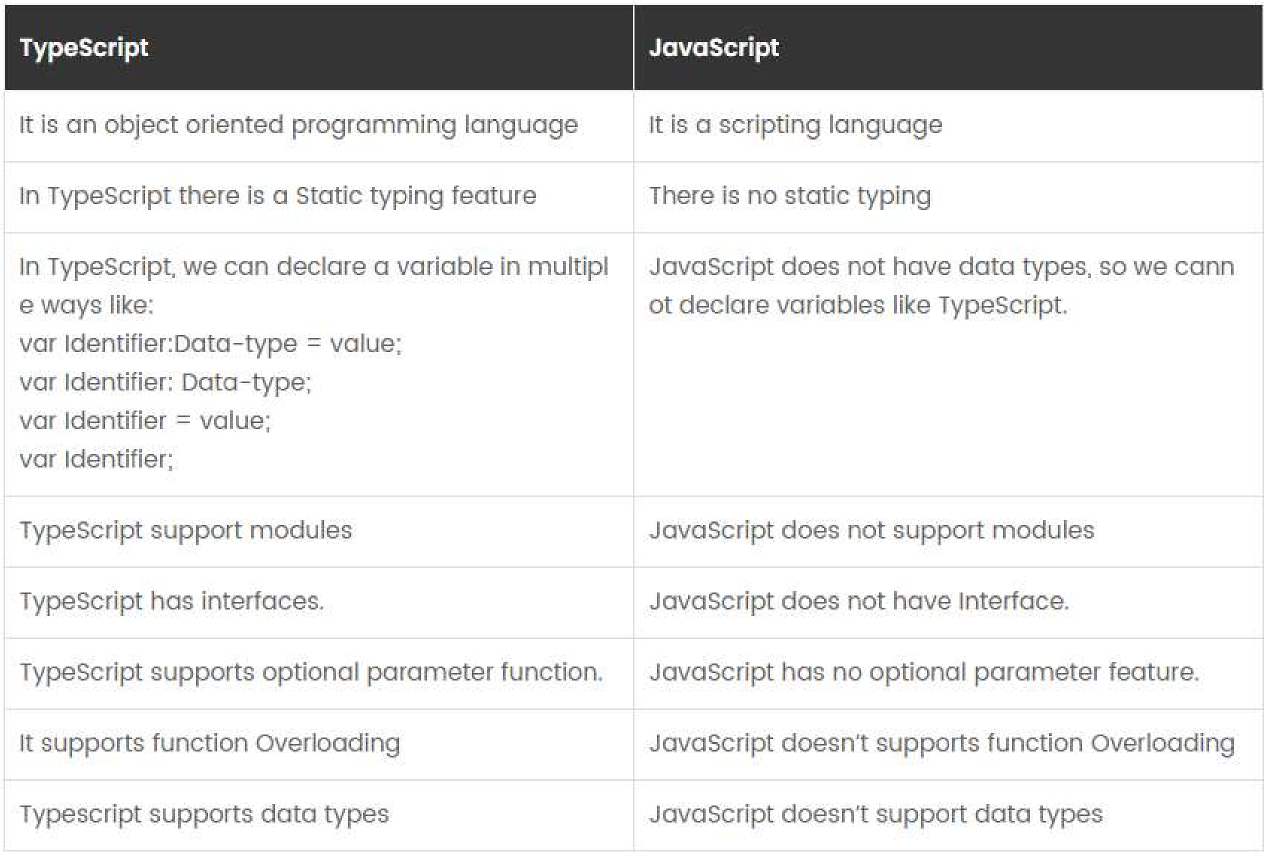


- In 2010 **Anders Hejlsberg** (**the founder of Typescript**) started working on Typescript at Microsoft and in 2012 the first version of Typescript was released to the public (Typescript 0.8).

Why should we use TypeScript?

- TypeScript simplifies JavaScript code which is easier to read and debug
- TypeScript is Open source
- TypeScript provides highly productive development tools for JavaScript IDE and practices like static checking
- TypeScript makes code easier to read and understand
- With TypeScript, we can make a huge improvement over plain JavaScript
- TypeScript gives us all the benefits of ES6 (ECMAScript 6), plus more productivity
- TypeScript can help to avoid painful bugs that developers commonly run into when writing JavaScript by type-checking the code
- The powerful type system, including generics
- TypeScript is nothing but JavaScript with some additional features
- Structural, rather than nominal
- TypeScript code can be compiled as per ES5 and ES6 standards to support the latest browser
- Aligned with ECMAScript for compatibility
- Starts and ends with JavaScript
- Supports static typing
- TypeScript will save your developers time
- TypeScript is a superset of ES3, ES5, and ES6

Differences between TypeScript and JavaScript



TypeScript	JavaScript
It is an object oriented programming language	It is a scripting language
In TypeScript there is a Static typing feature	There is no static typing

In TypeScript, we can declare a variable in multipl
e ways like:
var Identifier:Data-type = value;
var Identifier: Data-type;
var Identifier = value;
var Identifier;
	JavaScript does not have data types, so we cann
	ot declare variables like TypeScript.

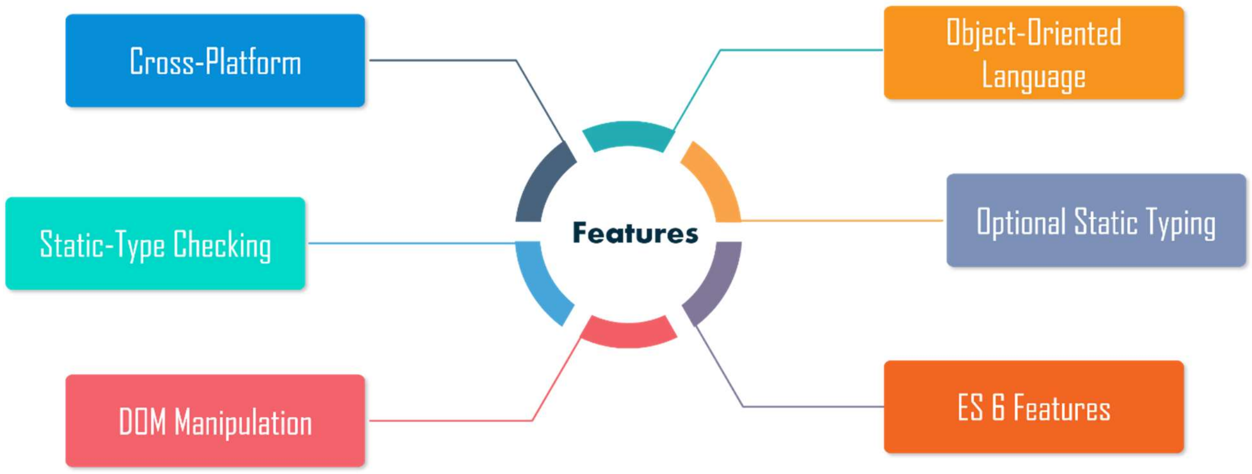
TypeScript support modules	JavaScript does not support modules
TypeScript has interfaces.	JavaScript does not have Interface.
TypeScript supports optional parameter function.	JavaScript has no optional parameter feature.
It supports function Overloading	JavaScript doesn't supports function Overloading
Typescript supports data types	JavaScript doesn't support data types

TypeScript Features

- **Cross-Platform:** TypeScript runs on any platform that JavaScript runs on. The TypeScript compiler can be installed on any Operating System such as Windows, macOS, and Linux.
- **Object-Oriented Language:** TypeScript provides powerful features such as Classes, Interfaces, and Modules. You can write pure object-oriented code for client-side as well as server-side development.
- **Static type-checking:** TypeScript uses static typing. This is done using type annotations. It helps type checking at compile time. Thus, you can find errors while typing the code without running your script each time. Additionally, using the type

inference mechanism, if a variable is declared without a type, it will be inferred based on its value.

- **Optional Static Typing:** TypeScript static typing is optional, if you prefer to use JavaScript's dynamic typing.
- **DOM Manipulation:** Like JavaScript, TypeScript can be used to manipulate the DOM.
- **ES 6 Features:** TypeScript includes most features of planned ECMAScript 2015 (ES 6, 7) such as class, interface, Arrow functions etc.



Power of Types

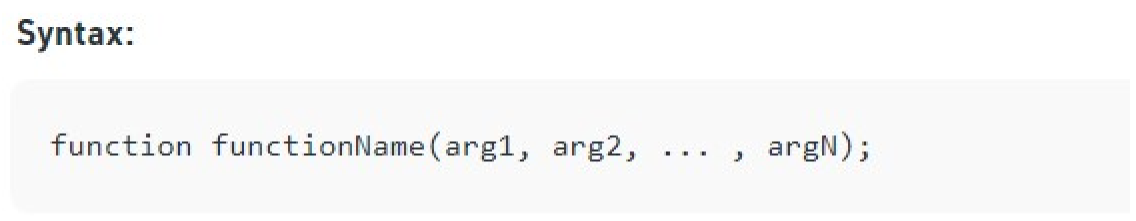
TypeScript is a JavaScript-based programming language with a typed syntax. It provides improved tools of any size. It adds extra syntax to JavaScript. This helps in facilitating a stronger interaction between you and your editor. It also helps in catching the mistakes well in advance.

It uses type inference to provide powerful tools without the need for extra code. TypeScript may be executed everywhere JavaScript is supported as it can be converted to JavaScript code.

TypeScript Functions?

Functions are pieces of code that execute specified tasks. They are used to implement object-oriented programming principles like classes, objects, polymorphism, and abstraction. It is used to ensure the reusability and maintainability of the program. Although TypeScript has the idea of classes and modules, functions are still an important aspect of the language.

Function declaration: The name, parameters, and return type of a function are all specified in a function declaration. The function declaration has the following:



Syntax:

```
function functionName(arg1, arg2, ... , argN);
```

Function definition: It includes the actual statements that will be executed. It outlines what should be done and how it should be done. The function definition has the following

Function call: A function can be called from anywhere in the application. In both function calling and function definition, the parameter/argument must be the same. We must pass the same number of parameters that the function definition specifies.

Types of Functions in TypeScript: There are two types of functions

- Named Function
- Anonymous Function

1. Named function: A named function is defined as one that is declared and called by its given name. They may include parameters and have return types.

Syntax:

```
functionName( [args] ) { }
```

```
// Named Function Definition
function myFunction(x: number, y: number): number {
  return x + y;
}

// Function Call
myFunction(7, 5);
```

2. Anonymous Function: An anonymous function is a function without a name. At runtime, these kinds of functions are dynamically defined as an expression. We may save it in a variable and eliminate the requirement for function names. They accept inputs and return outputs in the same way as normal functions do. We may use the variable name to call it when we need it. The functions themselves are contained inside the variable.

Syntax:

```
let result = function( [args] ) { }
```

```
// Anonymous Function
let myFunction = function (a: number, b: number) : number {
    return a + b;
};

// Anonymous Function Call
console.log(myFuction(7, 5));
```

Advantage of function: Benefits of functions may include but are not limited to the following:

- **Code Reusability**: We can call a function several times without having to rewrite the same code block. The reusability of code saves time and decreases the size of the program.

- **Less coding**: Our software is more concise because of the functions. As a result, we don't need to write a large number of lines of code each time we execute a routine activity.

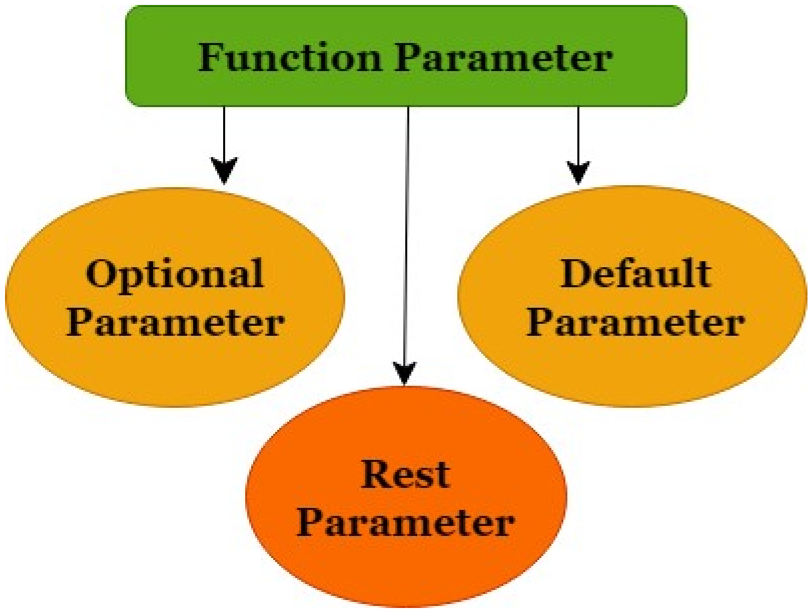
- **Easy to debug:** It makes it simple for the programmer to discover and isolate incorrect data.

TypeScript Function Parameter

Functions are the basic building block of any application which holds some business logic. The process of creating a function in TypeScript is similar to the process in JavaScript.

In functions, parameters are the values or arguments that passed to a function. The TypeScript, compiler accepts the same number and type of arguments as defined in the function signature. If the compiler does not match the same parameter as in the function signature, then it will give the compilation error.

1. Optional Parameter
2. Default Parameter
3. Rest Parameter



Optional Parameter:

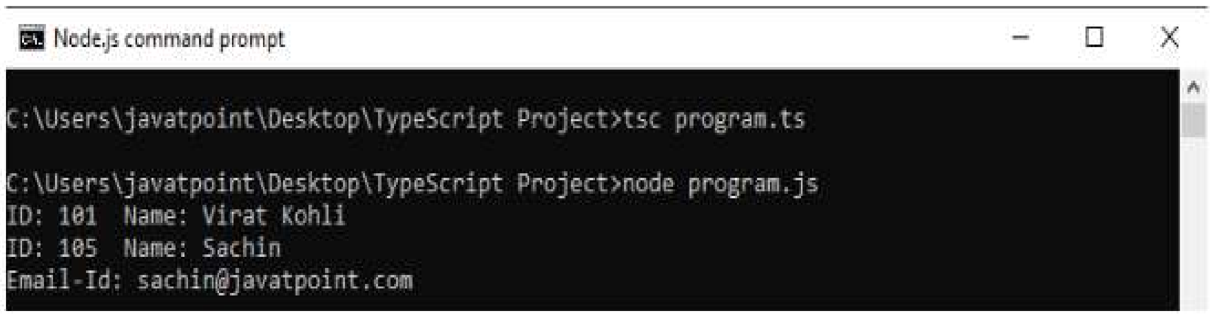
JavaScript allows us to call a function without passing any arguments. Hence, in a JavaScript function, the parameter is optional. If we declare the function without passing arguments, then each parameter value is undefined.

Unlike JavaScript, the TypeScript compiler will throw an error if we try to invoke a function without providing the exact number and types of parameters as declared in its function signature. To overcome this problem, TypeScript introduces **optional parameter**. We can use optional parameters by using the **question mark sign ('?')**. It means that the parameters which may or may not receive a value can be appended with a "?" sign to mark them as optional. In the below example, **e_mail_id** is marked as an optional parameter.

Example

```
function showDetails(id:number,name:string,e_mail_id?:string) {
   console.log("ID:", id, " Name:",name);
   if(e_mail_id!=undefined)
   console.log("Email-Id:",e_mail_id);
}
showDetails(101,"Virat Kohli");
showDetails(105,"Sachin","sachin@javatpoint.com");
```

Output:



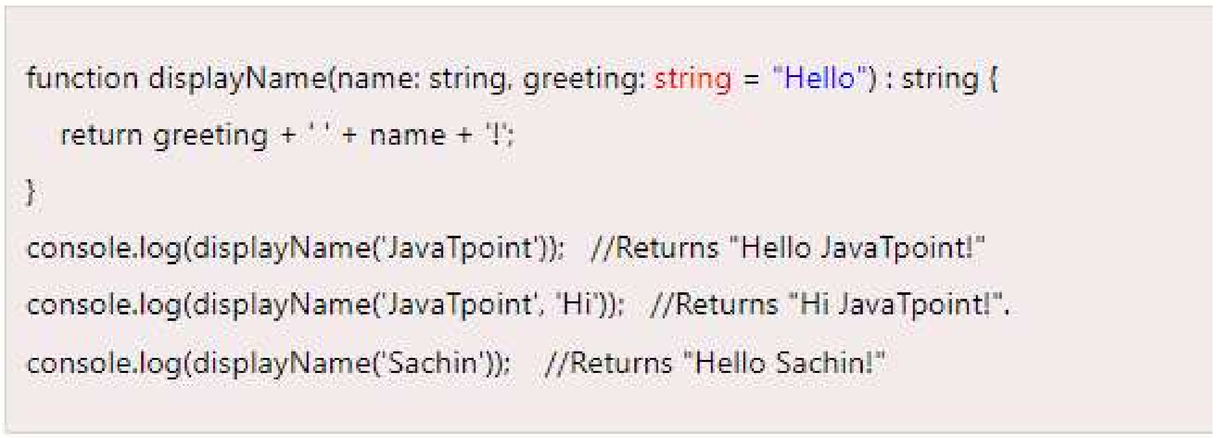
```
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts

C:\Users\javatpoint\Desktop\TypeScript Project>node program.js
ID: 101  Name: Virat Kohli
ID: 105  Name: Sachin
Email-Id: sachin@javatpoint.com
```

Default Parameter:

TypeScript provides an option to set default values to the function parameters. If the user does not pass a value to an argument, TypeScript initializes the default value for the parameter. The behavior of the default parameter is the same as an optional parameter. For the default parameter, if a value is not passed in a function call, then the default parameter must follow the required parameters in the function signature. However, if a function signature has a default parameter before a required parameter, we can still call a function which marks the default parameter is passed as an undefined value.

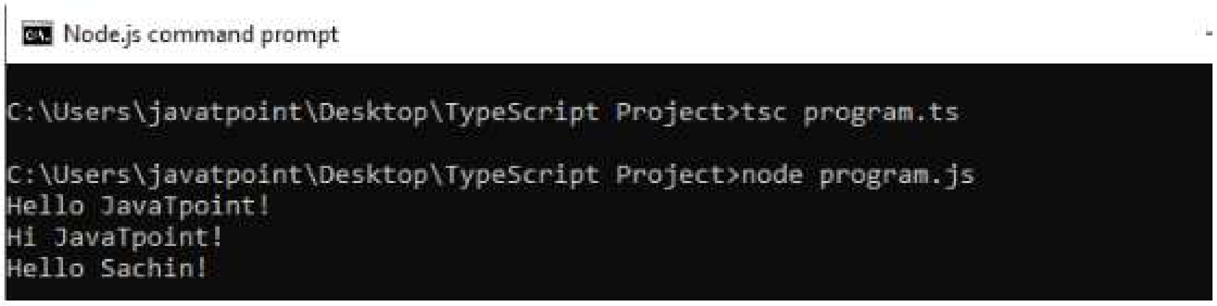
Note: We cannot make the parameter default and optional at the same time.

Example



```
function displayName(name: string, greeting: string = "Hello") : string {
    return greeting + ' ' + name + '!';
}
console.log(displayName('JavaTpoint'));   //Returns "Hello JavaTpoint!"
console.log(displayName('JavaTpoint', 'Hi'));   //Returns "Hi JavaTpoint!".
console.log(displayName('Sachin'));   //Returns "Hello Sachin!"
```

Output:



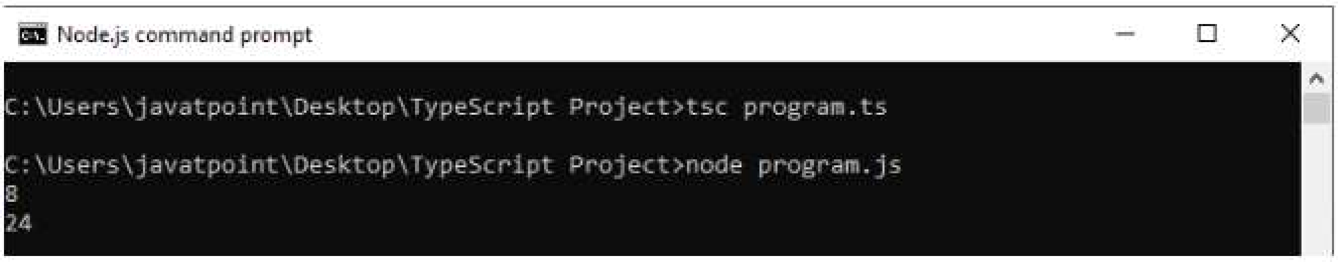
```
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts

C:\Users\javatpoint\Desktop\TypeScript Project>node program.js
Hello JavaTpoint!
Hi JavaTpoint!
Hello Sachin!
```

Rest Parameter:

It is used to pass **zero or more** values to a function. We can declare it by prefixing the **three "dot"** characters ('...') before the parameter. It allows the functions to have a different number of arguments without using the arguments object. The TypeScript compiler will create an array of arguments with the rest parameter so that all array methods can work with the rest parameter. The rest parameter is useful, where we have an undetermined number of parameters.

Example

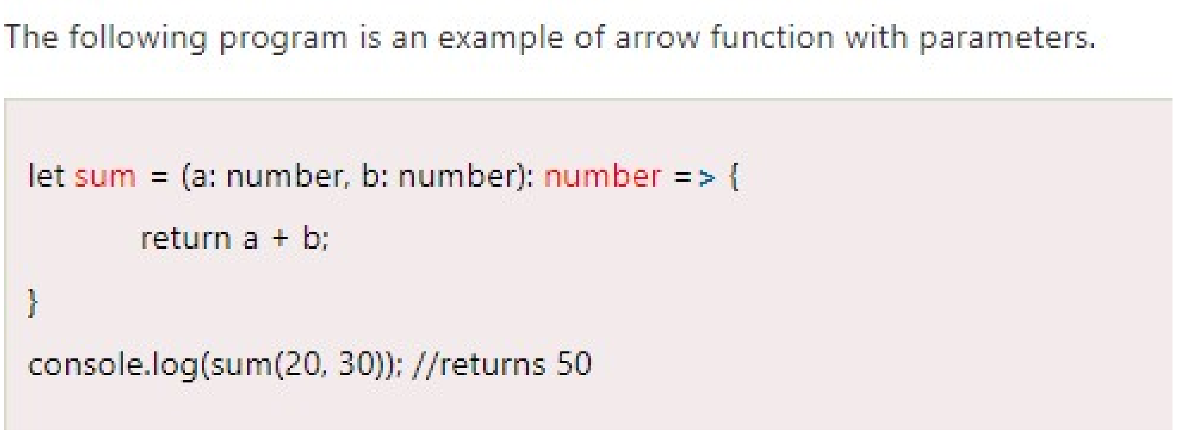
```
function sum(a: number, ...b: number[]): number {
  let result = a;
  for (var i = 0; i < b.length; i++) {
  result += b[i];
  }
  return result;
}
let result1 = sum(3, 5);
let result2 = sum(3, 5, 7, 9);
console.log(result1 +"\n" + result2);
```



```
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts

C:\Users\javatpoint\Desktop\TypeScript Project>node program.js
8
24
```

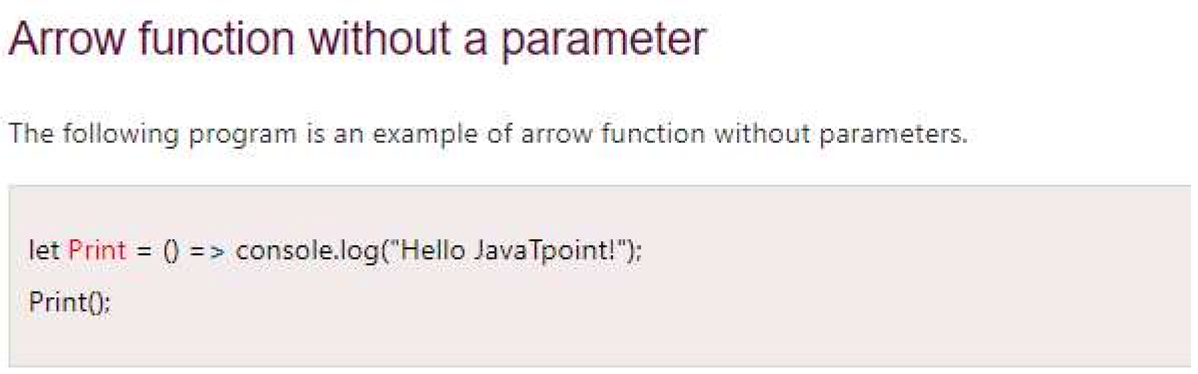
Arrow functions –

ES6 version of TypeScript provides an arrow function which is the **shorthand** syntax for defining the anonymous function, i.e., for function expressions. It omits the function keyword. We can call it fat arrow (because -> is a thin arrow and => is a "**fat**" arrow). It is also called a **Lambda function**. The arrow function has lexical scoping of "**this**" keyword.



The following program is an example of arrow function with parameters.

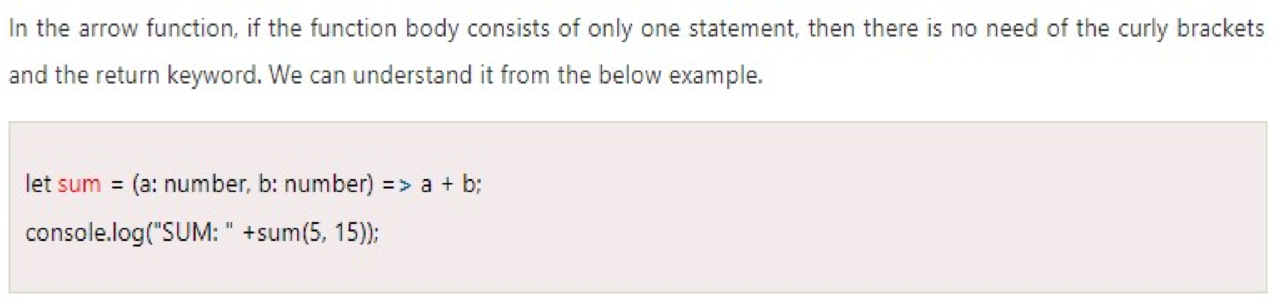
```
let sum = (a: number, b: number): number => {
        return a + b;
}
console.log(sum(20, 30)); //returns 50
```



Arrow function without a parameter

The following program is an example of arrow function without parameters.

```
let Print = () => console.log("Hello JavaTpoint!");
Print();
```



In the arrow function, if the function body consists of only one statement, then there is no need of the curly brackets and the return keyword. We can understand it from the below example.

```
let sum = (a: number, b: number) => a + b;
console.log("SUM: " +sum(5, 15));
```

Fat arrow notations are used for anonymous functions i.e for function expressions. They are also called lambda functions in other languages.

Syntax:

```
(param1, param2, ..., paramN) => expression
```

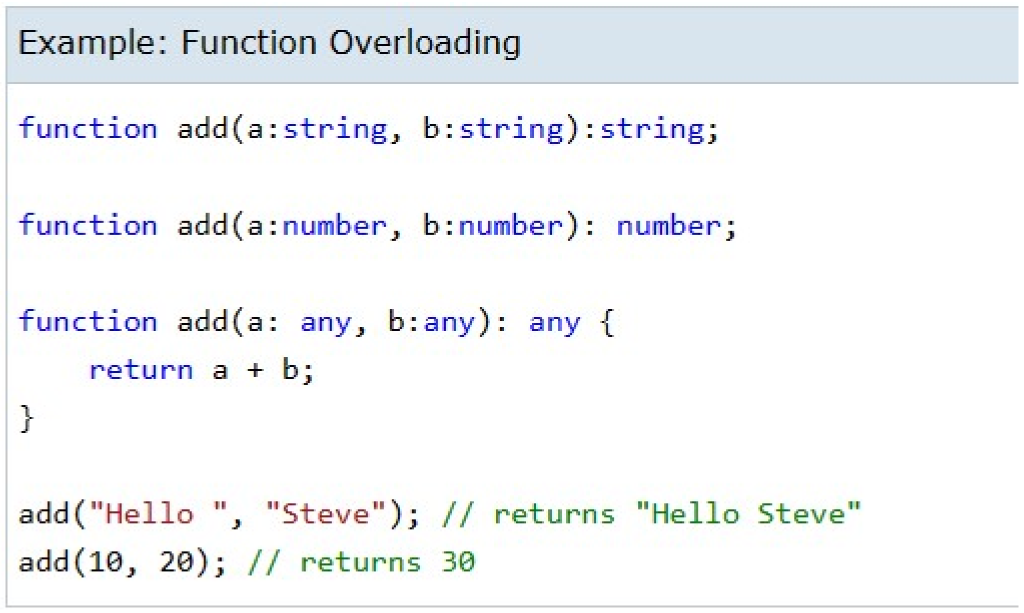
Using fat arrow =>, we dropped the need to use the function keyword. Parameters are passed in the parenthesis (), and the function expression is enclosed within the curly brackets { }.

Example: Fat Arrow Function

```
let sum = (x: number, y: number): number => {
    return x + y;
}

sum(10, 20); //returns 30
```

Function overloading

TypeScript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.



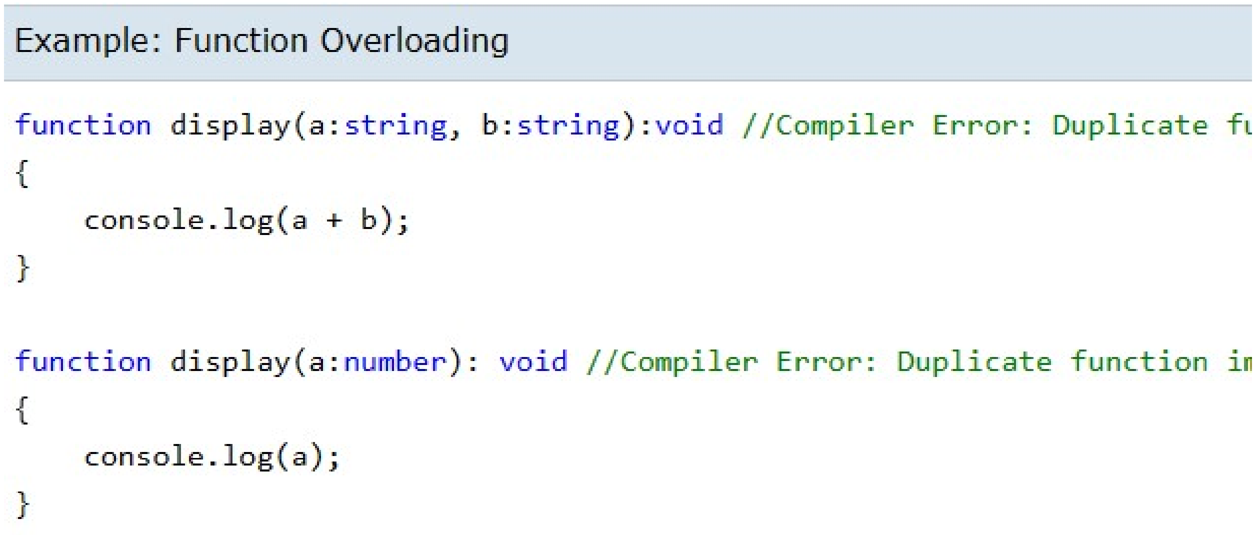
Example: Function Overloading

```
function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {
    return a + b;
}

add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30
```

In the above example, we have the same function `add ()` with two function declarations and one function implementation. The first signature has two parameters of type string, whereas the second signature has two parameters of the type number.

The last function should have the function implementation. Since the return type can be either string or number as per the first two function declarations, we must use compatible parameters and return type as `any` in the function definition.

Function overloading with different number of parameters and types with same name is not supported.


Example: Function Overloading

```
function display(a:string, b:string):void //Compiler Error: Duplicate fu
{
    console.log(a + b);
}

function display(a:number): void //Compiler Error: Duplicate function in
{
    console.log(a);
}
```

Thus, in order to achieve function overloading, we must declare all the functions with possible signatures. Also, function implementation should have compatible types for all declarations.

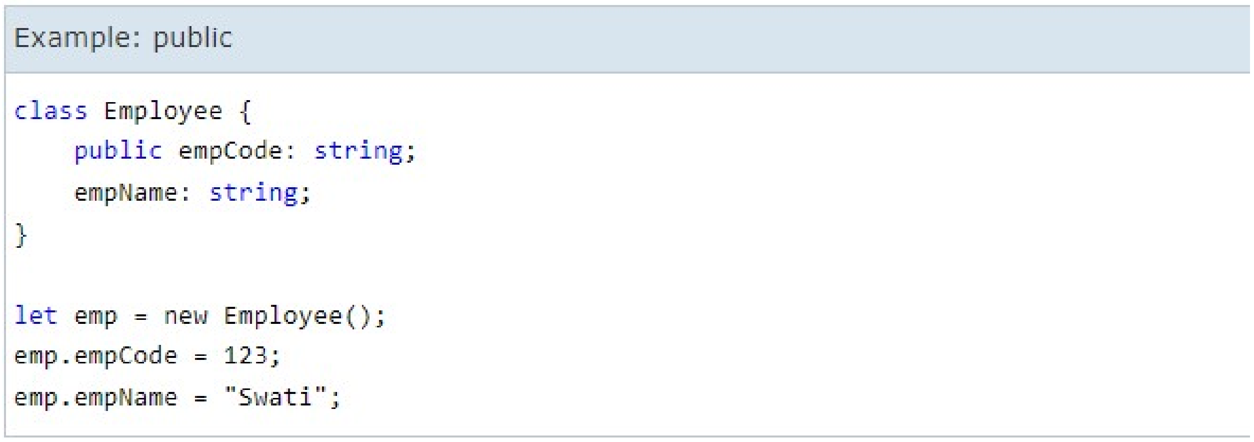
Access modifiers?

In object-oriented programming, the concept of 'Encapsulation' is used to make class members public or private i.e. a class can control the visibility of its data members. This is done using access modifiers. Type Script provides three access modifiers:

- private
- protected
- public

Public

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


Example: public

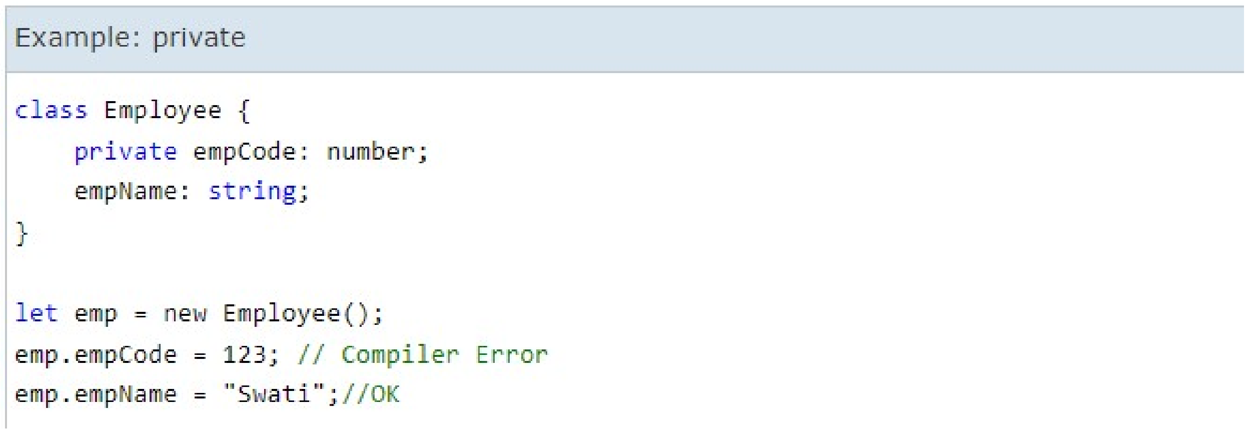
```
class Employee {
    public empCode: string;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123;
emp.empName = "Swati";
```

In the above example, **empCode** and **empName** are declared as public. So, they can be accessible outside of the class using an object of the class.

Please notice that there is not any modifier applied before **empName,** as TypeScript treats properties and methods as public by default if no modifier is applied to them.

Private

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

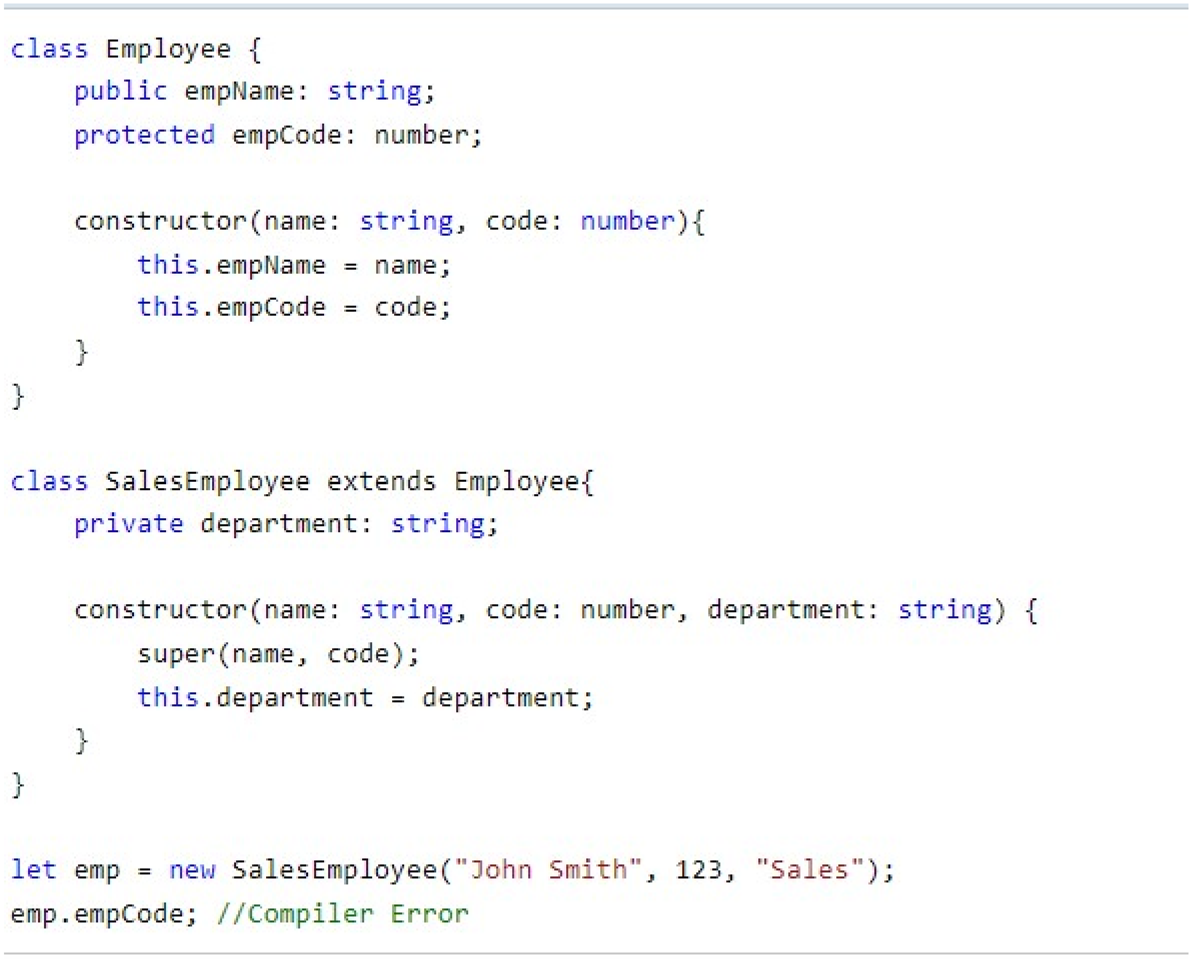

Example: private

```
class Employee {
    private empCode: number;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123; // Compiler Error
emp.empName = "Swati";//OK
```

In the above example, we have marked the member **empCode** as private. Hence, when we create an object **emp** and try to access the emp. empCode member, it will give an error.

Protected

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



```
class Employee {
    public empName: string;
    protected empCode: number;

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

class SalesEmployee extends Employee{
    private department: string;

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

let emp = new SalesEmployee("John Smith", 123, "Sales");
emp.empCode; //Compiler Error
```

In the above example, we have a class Employee with two members, public empName and protected property empCode. We create a subclass SalesEmployee that extends from the parent class Employee. If we try to access the protected member from outside the class, as emp.empCode, we get the following compilation error:

error TS2445: Property 'empCode' is protected and only accessible within class 'Employee' and its subclasses.

In addition to the access modifiers, TypeScript provides two more keywords: readOnly and static.

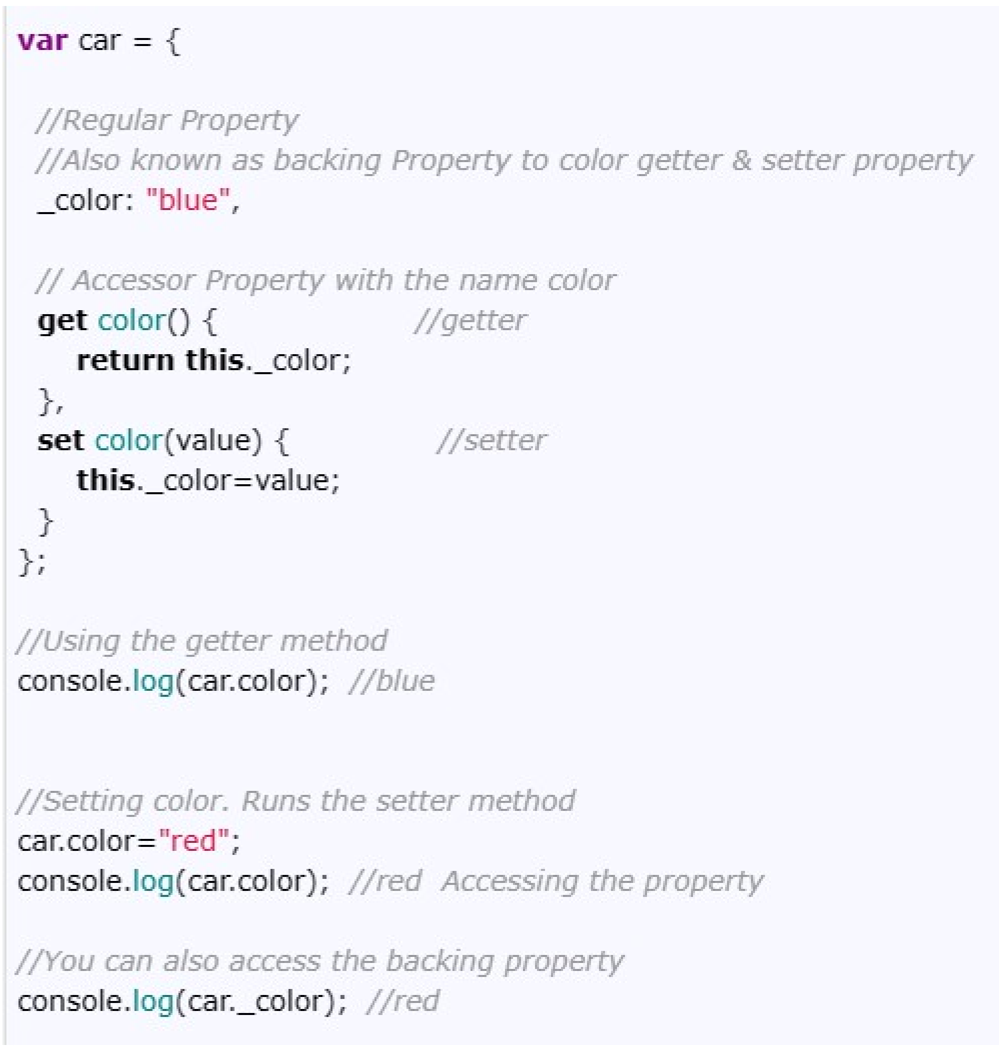
Summary of Access modifiers

TypeScript provides three access modifiers to class properties and methods: private, protected, and public.

- The private modifier allows access within the same class.
- The protected modifier allows access within the same class and subclasses.
- The public modifier allows access from any location.

Getters and setters?

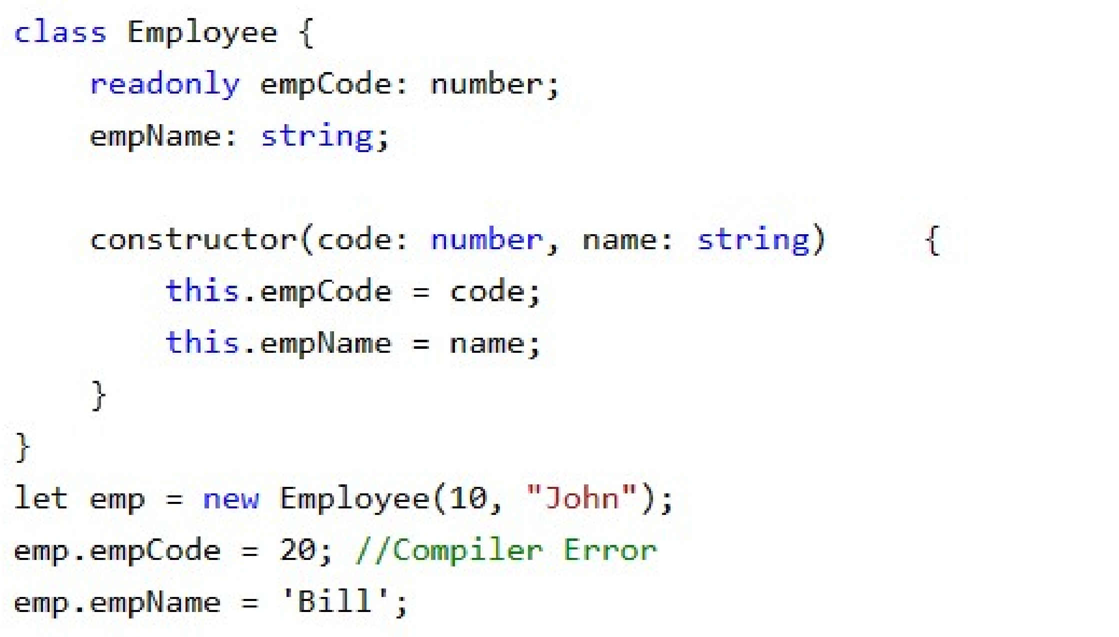
The Getters and Setters are known as accessor properties in TypeScript. They look like normal properties but are actually functions mapped to a Property. We use "**get**" to define a **getter** method and "**set**" to define a **setter method**. The Setter method runs when we assign a value to the Property. The Getter method runs when we access the Property.


```
var car = {

  //Regular Property
  //Also known as backing Property to color getter & setter property
  _color: "blue",

  // Accessor Property with the name color
  get color() {                //getter
     return this._color;
  },
  set color(value) {             //setter
     this._color=value;
  }
};

//Using the getter method
console.log(car.color);  //blue


//Setting color. Runs the setter method
car.color="red";
console.log(car.color);  //red  Accessing the property

//You can also access the backing property
console.log(car._color);  //red
```

Read-only

TypeScript includes the *readonly* keyword that makes a property as read-only in the class, type or interface.

Prefix readonly is used to make a property as read-only. 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 Employee {
    readonly empCode: number;
    empName: string;

    constructor(code: number, name: string)     {
        this.empCode = code;
        this.empName = name;
    }
}
let emp = new Employee(10, "John");
emp.empCode = 20; //Compiler Error
emp.empName = 'Bill';
```

In the above example, we have the Employee class with two properties- empennage and empCode. Since empCode is read only, it can be initialized at the time of declaration or in the constructor.

If we try to change the value of empCode after the object has been initialized, the compiler shows the following compilation error:

error TS2540: Cannot assign to empCode' because it is a constant or a read-only property.

Static?

The static members of a class are accessed using the class name and dot notation, without creating an object e.g. <ClassName>.<StaticMember>.

The static members can be defined by using the keyword *static*. Consider the following example of a class with static property.

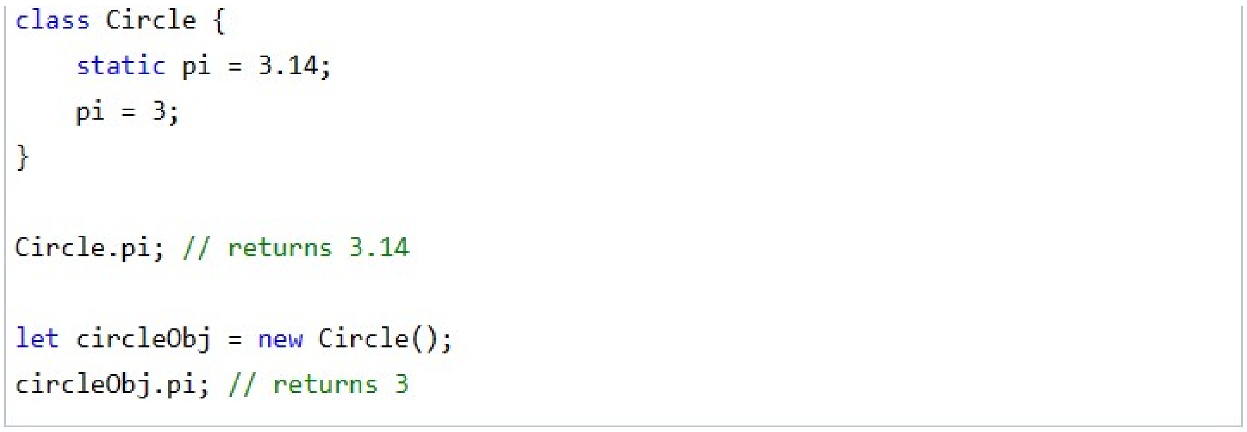
```
class Circle {
    static pi: number = 3.14;
}
```

The above `Circle` class includes a static property `pi`. This can be accessed using `Circle.pi`. TypeScript will generate the following JavaScript code for the above `Circle` class.

```
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
```

The above `Circle` class includes a static property and a static method. Inside the static method `calculateArea`, the static property can be accessed using this keyword or using the class name `Circle.pi`.

Now, consider the following example with static and non-static members.



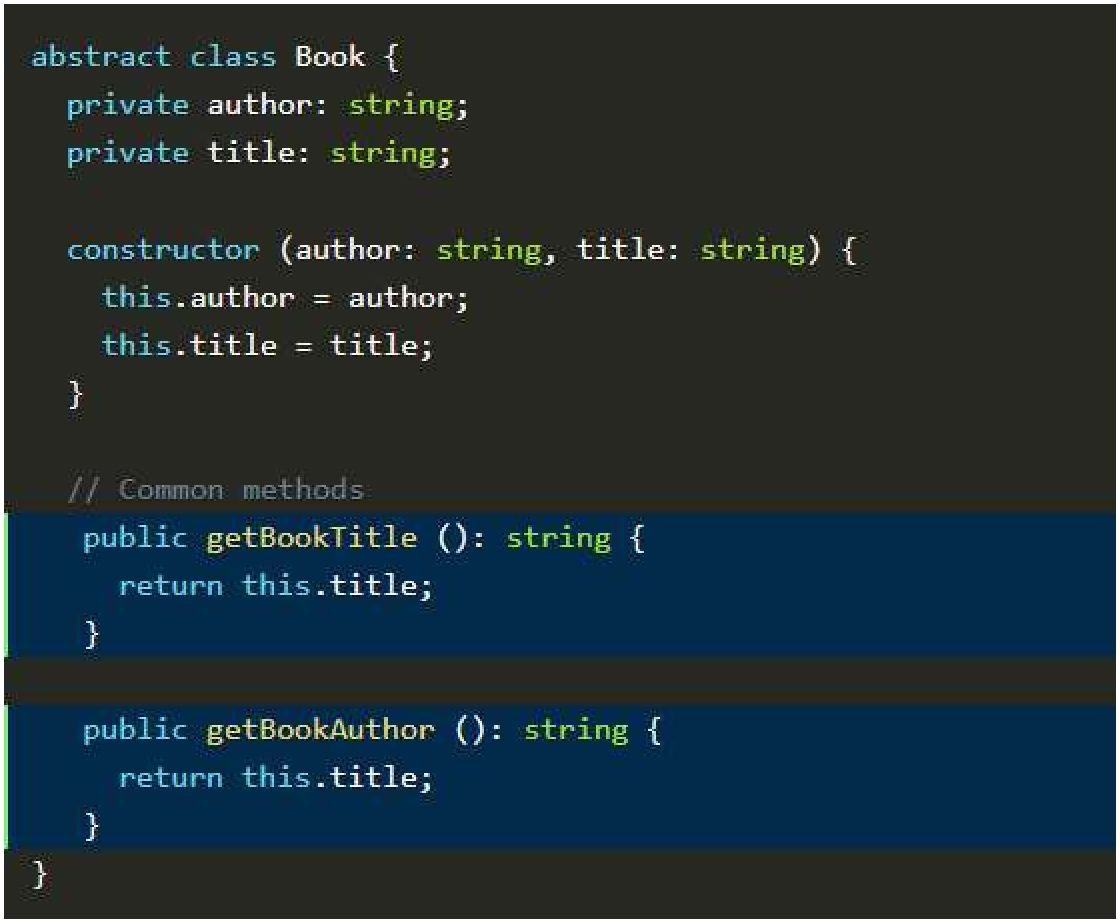
```
class Circle {
    static pi = 3.14;
    pi = 3;
}

Circle.pi; // returns 3.14

let circleObj = new Circle();
circleObj.pi; // returns 3
```

As you can see, static and non-static fields with the same name can exists without any error. The static field will be accessed using dot notation and the non-static field can be accessed using an object.

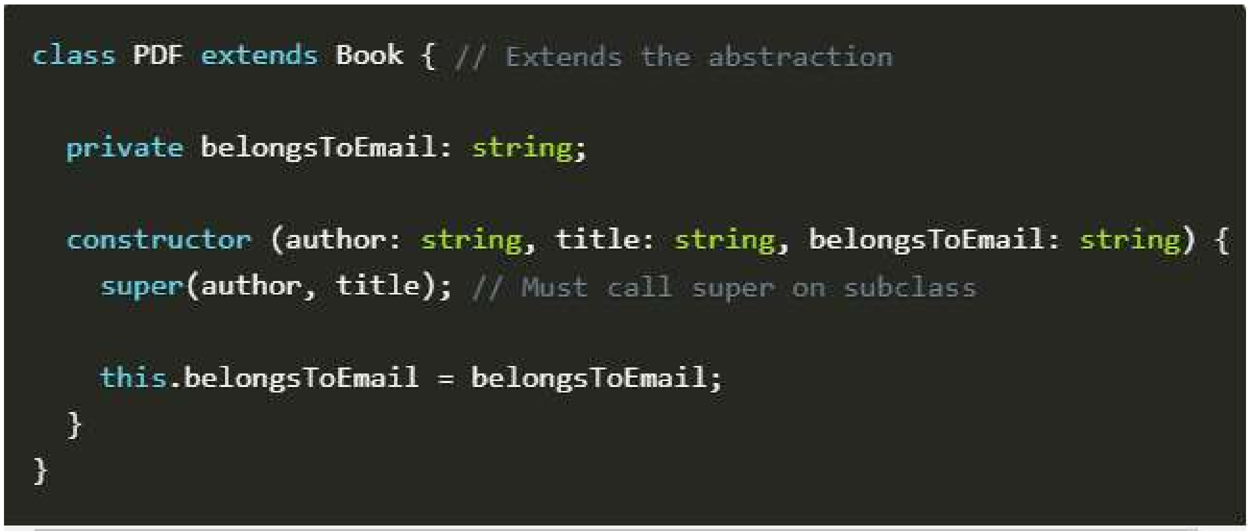
Abstract classes

Abstraction is one of the key ideas from object-oriented programming, and abstract classes are one of the tools that OO provides to implement abstraction.

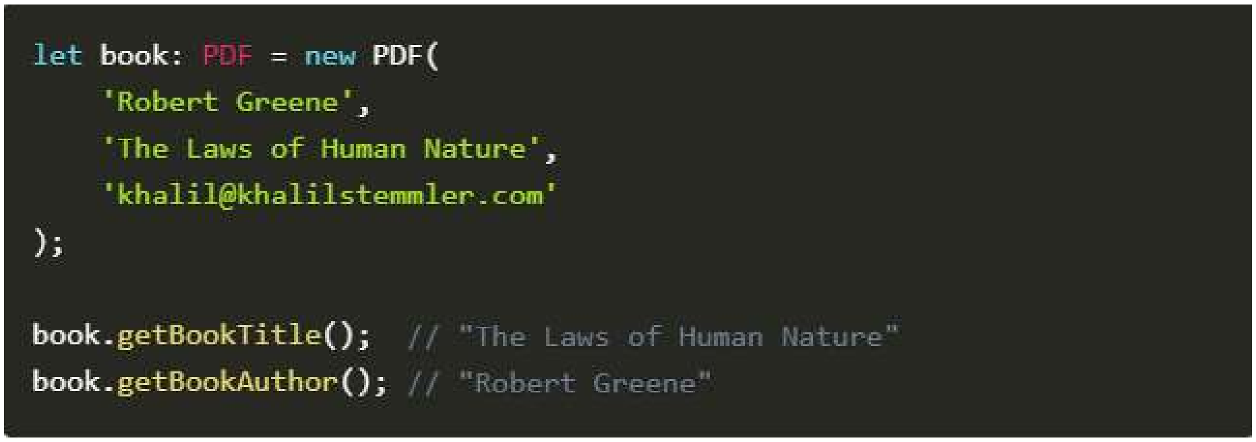
The most common use of abstract classes in TypeScript is to locate some common behavior to share within related subclasses.



```typescript
abstract class Book {
  private author: string;
  private title: string;

  constructor (author: string, title: string) {
    this.author = author;
    this.title = title;
  }

  // Common methods
  public getBookTitle (): string {
    return this.title;
  }

  public getBookAuthor (): string {
    return this.title;
  }
}
```



```typescript
class PDF extends Book { // Extends the abstraction

  private belongsToEmail: string;

  constructor (author: string, title: string, belongsToEmail: string) {
    super(author, title); // Must call super on subclass

    this.belongsToEmail = belongsToEmail;
  }
}
```



```
let book: PDF = new PDF(
    'Robert Greene',
    'The Laws of Human Nature',
    'khalil@khalilstemmler.com'
);

book.getBookTitle();  // "The Laws of Human Nature"
book.getBookAuthor(); // "Robert Greene"
```

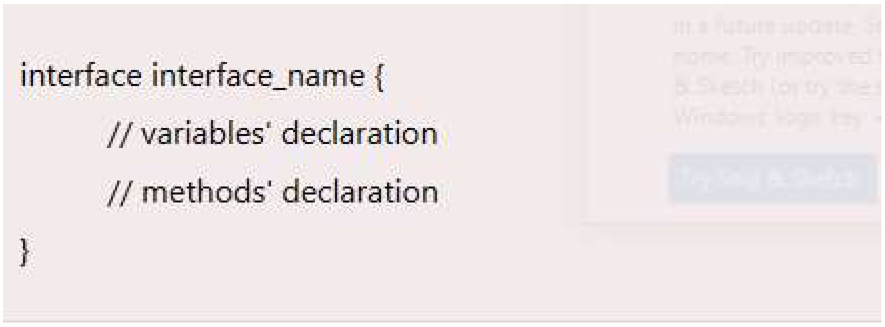
Interfaces?

An Interface is a structure which acts as a **contract** in our application. It defines the syntax for classes to follow, means a class which implements an interface is bound to implement all its members. We cannot instantiate the interface, but it can be referenced by the class object that implements it. The TypeScript compiler uses interface for **type-checking** (also known as "duck typing" or "structural subtyping") whether the object has a specific structure or not.

The interface contains only the **declaration** of the **methods** and **fields**, but not the **implementation**. We cannot use it to build anything. It is inherited by a class, and the class which implements interface defines all members of the interface.

When the Typescript compiler compiles it into JavaScript, then the interface will be disappeared from the JavaScript file. Thus, its purpose is to help in the development stage only.

We can declare an interface as below.



```
interface interface_name {
        // variables' declaration
        // methods' declaration
}
```

- An **interface** is a keyword which is used to declare a TypeScript Interface.
- An **interface_name** is the name of the interface.
- An interface body contains variables and methods declarations.

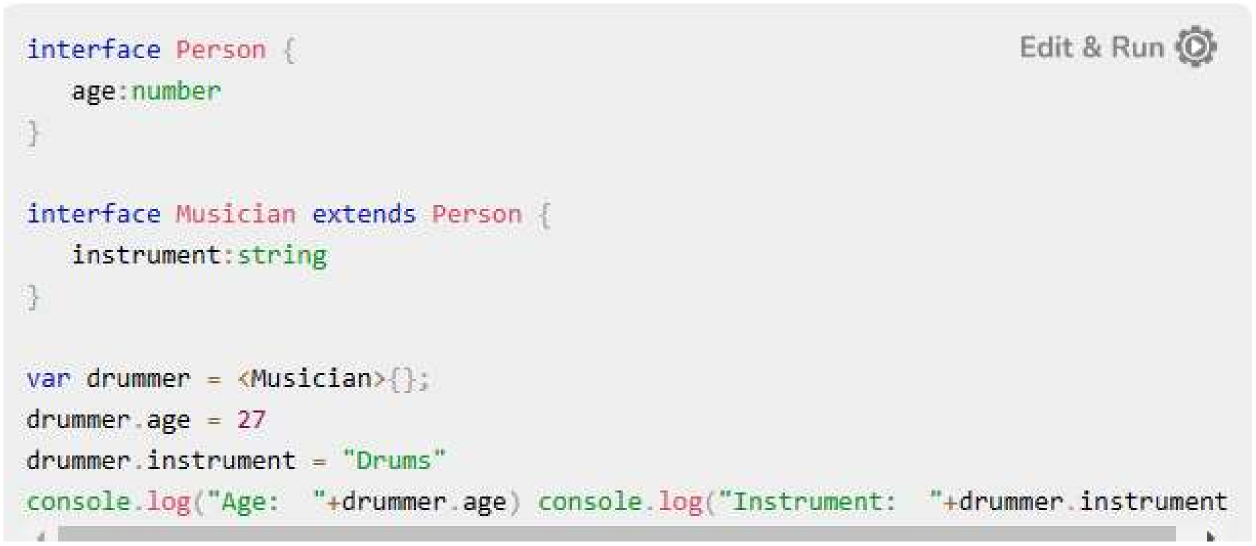
We can use the interface for the following things:

- Validating the specific structure of properties
- Objects passed as parameters
- Objects returned from functions.

TypeScript interface is an interface between specification and implementation. TypeScript interface does not contain any implementation. It is used for setting the standards for the implementation classes to do the implementation.

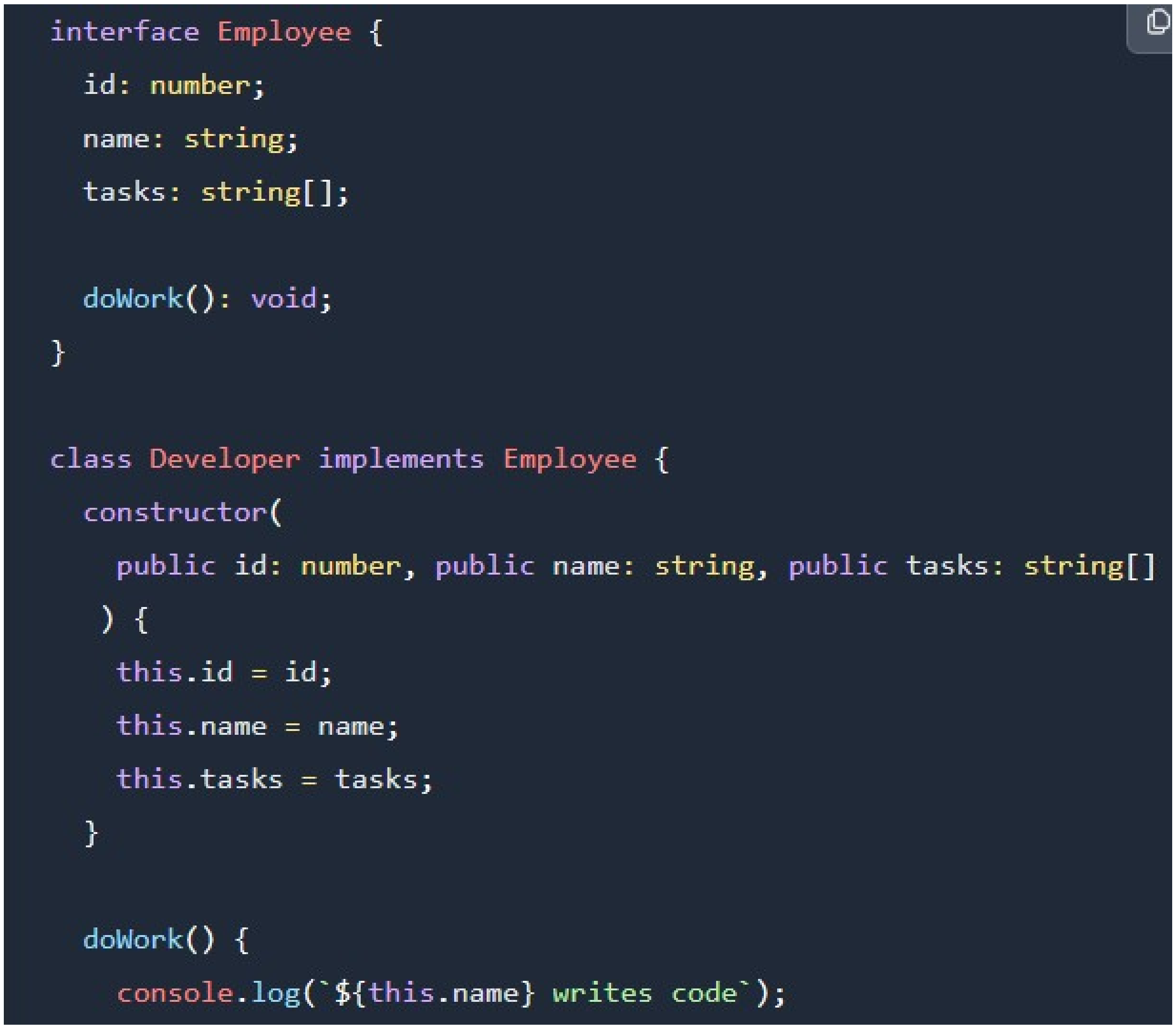
Extending and Implementing Interface?

We can use the "**extends**" keyword to implement inheritance among interfaces.



```
interface Person {
   age:number
}

interface Musician extends Person {
   instrument:string
}

var drummer = <Musician>{};
drummer.age = 27
drummer.instrument = "Drums"
console.log("Age:  "+drummer.age) console.log("Instrument:  "+drummer.instrument
```

TypeScript also allows us to use the interface in a class. A class implements the interface by using the **implements** keyword. We can understand it with the below example.



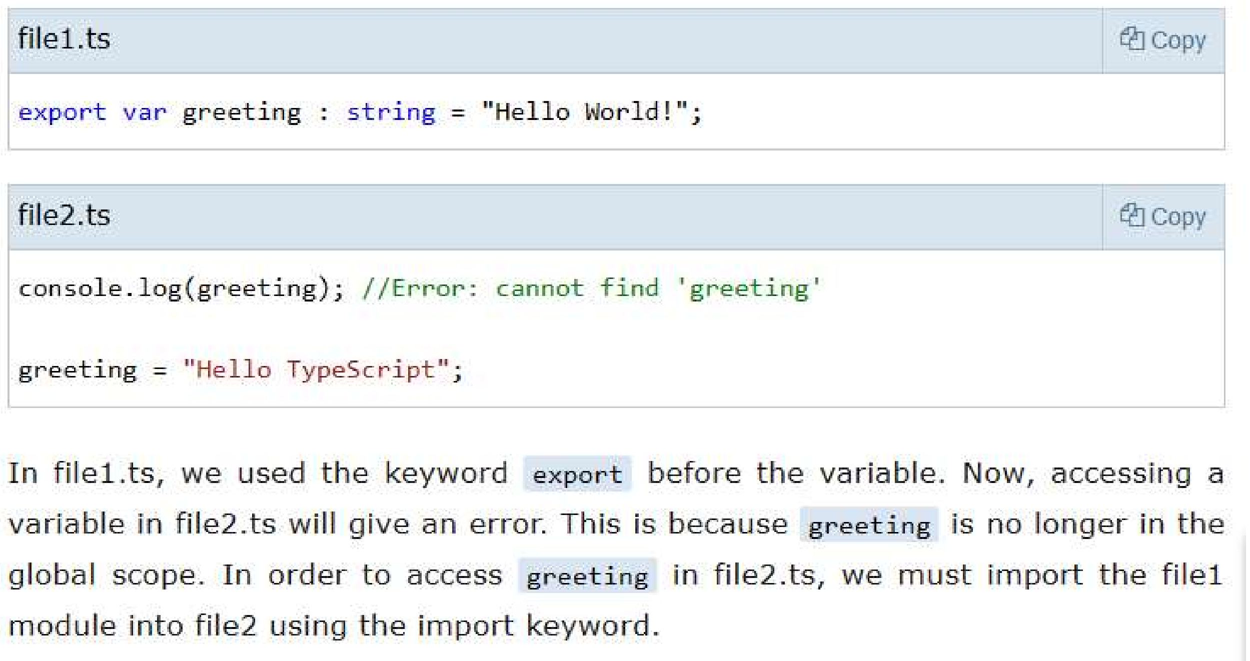
```
interface Employee {
  id: number;
  name: string;
  tasks: string[];

  doWork(): void;
}

class Developer implements Employee {
  constructor(
    public id: number, public name: string, public tasks: string[]
   ) {
    this.id = id;
    this.name = name;
    this.tasks = tasks;
  }

  doWork() {
    console.log(`${this.name} writes code`);
```

Import and Export modules?

TypeScript provides modules and namespaces in order to prevent the default global scope of the code and also to organize and maintain a large code base.

Modules are a way to create a local scope in the file. So, all variables, classes, functions, etc. that are declared in a module are not accessible outside the module. A module can be created using the keyword `export` and a module can be used in another module using the keyword `import`.

In TypeScript, files containing a top-level export or import are considered modules. For example, we can make the above files as modules as below.



file1.ts

```
export var greeting : string = "Hello World!";
```

file2.ts

```
console.log(greeting); //Error: cannot find 'greeting'

greeting = "Hello TypeScript";
```

In file1.ts, we used the keyword `export` before the variable. Now, accessing a variable in file2.ts will give an error. This is because `greeting` is no longer in the global scope. In order to access `greeting` in file2.ts, we must import the file1 module into file2 using the import keyword.

Export Module

A module can be defined in a separate .ts file which can contain functions, variables, interfaces and classes. Use the prefix export with all the definitions you want to include in a module and want to access from other modules.

Employee.ts

```
export let age : number = 20;
export class Employee {
    empCode: number;
    empName: string;
    constructor(name: string, code: number) {
        this.empName = name;
        this.empCode = code;
    }
    displayEmployee() {
        console.log ("Employee Code: " + this.empCode + ", Employee Name: " + thi
    }
}
let companyName:string = "XYZ";
```

In the above example, `Employee.ts` is a module which contains two variables and a class definition. The `age` variable and the `Employee` class are prefixed with the export keyword, whereas `companyName` variable is not. Thus, `Employee.ts` is a module which exports the `age` variable and the `Employee` class to be used in other modules by importing the `Employee` module using the import keyword. The `companyName` variable cannot be accessed outside this `Employee` module, as it is not exported.

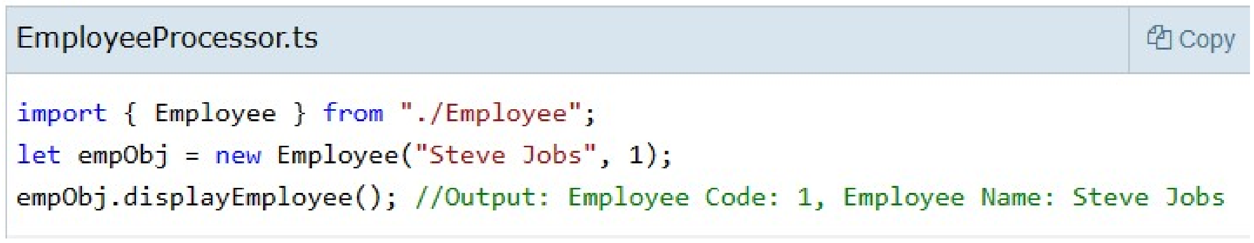
Import Module

A module can be used in another module using an import statement.

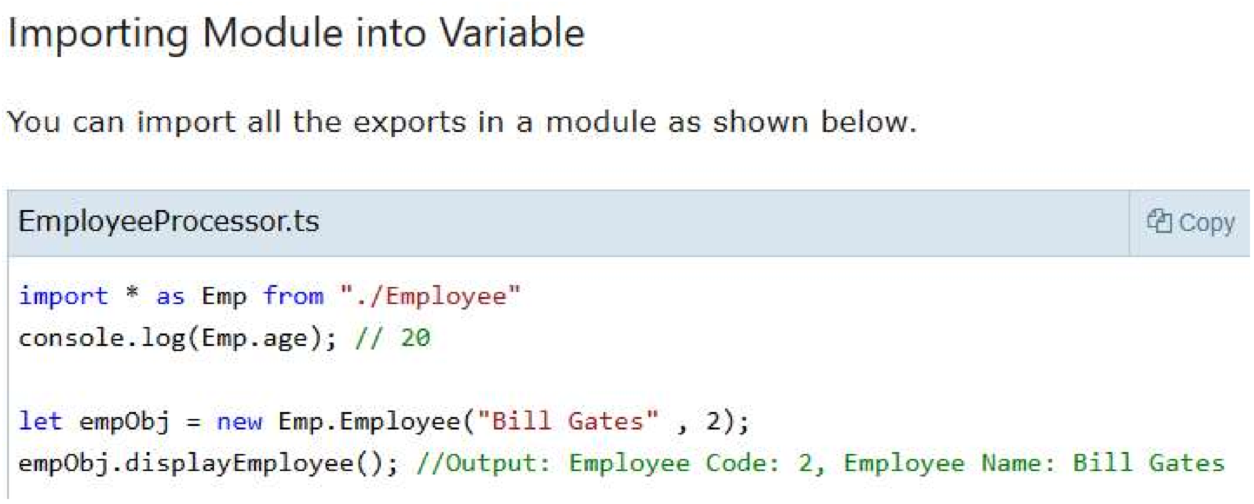
Syntax:

```
Import { export name } from "file path without extension"
```

We exported a variable and a class in the `Employee.ts`. However, we can only import the export module which we are going to use. The following code only imports the `Employee` class from `Employee.ts` into another module in the `EmployeeProcessor.ts` file.


EmployeeProcessor.ts

```
import { Employee } from "./Employee";
let empObj = new Employee("Steve Jobs", 1);
empObj.displayEmployee(); //Output: Employee Code: 1, Employee Name: Steve Jobs
```


Importing Module into Variable

You can import all the exports in a module as shown below.

EmployeeProcessor.ts

```
import * as Emp from "./Employee"
console.log(Emp.age); // 20

let empObj = new Emp.Employee("Bill Gates" , 2);
empObj.displayEmployee(); //Output: Employee Code: 2, Employee Name: Bill Gates
```

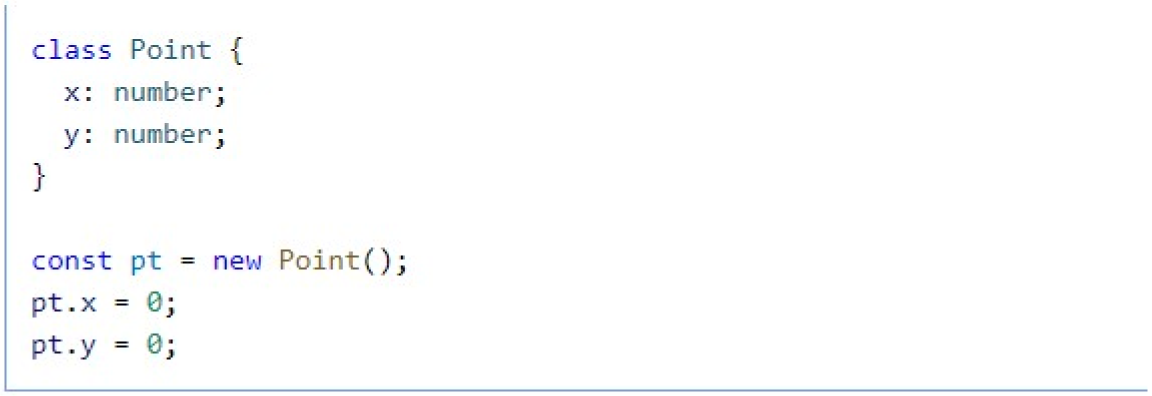
In the above example, we import all the exports in `Employee` module in a single variable called `Emp`. So, we don't need to write an export statement for each individual module. In the above example, it will import `age` and `Employee` class into the `Emp` variable and can be accessed using `Emp.age` and `Emp.Employee`.

Important notes –

1. The Typescript variables have three scopes. Global, Function and local.

2. Typescript comprises three main components: Language, the TypeScript Compiler, and the TypeScript Language Service.

3. TypeScript offers full support for the class keyword



```
class Point {
  x: number;
  y: number;
}

const pt = new Point();
pt.x = 0;
pt.y = 0;
```

Q1. What is TypeScript and why do we need it?

Ans. JavaScript is the only client side language universally supported by all browsers. But JavaScript is not the best designed language. It's not a class-based object-oriented language, doesn't support class based inheritance, unreliable dynamic typing and lacks in compile time error checking. And TypeScript addresses all these problems. In other words, TypeScript is an attempt to "fix" JavaScript problems.

TypeScript is a free and open source programming language developed and maintained by Microsoft. It is a strict superset of JavaScript, and adds **optional static typing** and **class-based object-oriented programming** to the language. TypeScript is quite easy to learn and use for developers familiar with C#, Java and all strong typed languages. At the end of day "TypeScript is a language that generates plain JavaScript files."

As stated on Typescript official website, "TypeScript lets you write JavaScript the way you really want to. TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open Source." Where "**typed**" means that it considers the types of variables, parameters and functions.

Q3. What are different components of TypeScript?

Ans. There are mainly 3 components of TypeScript .

1. **Language** – The most important part for developers is the new language. The language consist of new syntax, keywords and allows you to write TypeScript.
2. **Compiler** – The TypeScript compiler is open source, cross-platform and open specification, and is written in TypeScript. Compiler will compile your TypeScript into JavaScript. And it will also emit error, if any. It can also help in concating different files to single output file and in generating source maps.
3. **Language Service** – TypeScript language service which powers the interactive TypeScript experience in Visual Studio, VS Code, Sublime, the TypeScript playground and other editor.

Q6. Is it possible to combine multiple .ts files into a single .js file?

Ans. Yes, it possible. While compiling add `--outFILE [OutputJSFileName]` option.

```
tsc --outFile comman.js file1.ts file2.ts file3.ts
```

This will compile all 3 ".ts" file and output into single "comman.js" file. And what will happen if you don't provide a output file name.

```
tsc --outFile file1.ts file2.ts file3.ts
```

In this case, file2.ts and file3.ts will be compiled and the output will be placed in file1.ts. So now your file1.ts contains JavaScript code.

Q8. Does TypeScript support all object oriented principles?

Ans. The answer is **YES**. There are 4 main principles to Object Oriented Programming: Encapsulation, Inheritance, Abstraction, and Polymorphism. TypeScript can implement all four of them with its smaller and cleaner syntax. Read Write Object-Oriented JavaScript with TypeScript.

Q9. Which object oriented terms are supported by TypeScript?

Ans. TypeScript supports following object oriented terms.

- Modules
- Classes
- Interfaces
- Data Types
- Member functions

Q10. Which are the different data types supported by TypeScript?

Ans. TypeScript supports following data types.

- Boolean `var bValue: boolean = false;`
- Number `var age: number = 16;`
- String `var name: string = "jon";`
- Array `var list:number[] = [1, 2, 3];`
- Enum

```
enum Color {Red, Green, Blue};
var c: Color = Color.Green;
```

- Any `var unknownType: any = 4;`
- Void

```
function NoReturnType(): void {
}
```

Q11. How TypeScript is optionally statically typed language?

Ans. TypeScript is referred as optionally statically typed, which means you can ask the compiler to ignore the type of a variable. Using `any` data type, we can assign any type of value to the variable. TypeScript will not give any error checking during compilation.

```
var unknownType: any = 4;
unknownType = "Okay, I am a string";
unknownType = false; // A boolean.
```

Q12. What are modules in TypeScript?

Ans. Modules are the way to organize code in TypeScript. Modules don't have any features, but can contain classes and interfaces. It is same like namespace in C#.

Q13. What are classes in TypeScript?

Ans. The concept of classes is very similar to .Net/Java. A Class can have constructor, member variables, properties and methods. TypeScript also allows access modifiers "private" and "public" for member variables and functions.