1 TypeScript compiler

Typecript is a superset of Javascript which means any JavaScript code is considered valid TypeScript code.

Because browsers only know JavaScript, they look for .js files with JavaScript EC 5 code and not .ts files. For that we need the TypeScript compiler which is the tool that transforms .ts files into .js files with JavaScript syntax for browsers to understand and execute.

The TypeScript compiler, named **tsc**, is written in TypeScript and can be used in any JavaScript host.

The most important TypeScript compiler command is the compilation command. This is the basic form for this command.

tsc myscript.ts

If no errors are found during compilation, the output of this command is a JavaScript file with the same name but with a .js extension (in this case it is going to be myscript.js)

tsc myscript.ts –noImplicitAny noImplicitAny tells the compiler to check the TypeScript code and raise an error on expressions and declarations with an implied any type.

tsc myscript.ts –target "ES5" target option requires

2 Using types

```
// in JavaScript
function add(x, y) { return x + y;}
```

```
// in TypeScript
function add(x: number, y: number) { return x + y;}
```

```
// in TypeScript
function add(x: string, y: string) { return x + y;}
```

3 Declare variable in TypeScript

let and const are two relatively new types of variable declarations in JavaScript which TypeScript supports.

using let keyword

```
let variableName: variableType = value;
// or
let variableName: variableType;
```

example:

```
let x: number = 5;
// or
let x: number;
```

let is a keyword which tells the compiler here comes a new declaration of a variable. let is similar to var in JavaScript.

; is optional in both JavaScript and TypeScript.

using const keyword: const declarations are another way of declaring variables. They are like let declarations but, as their name implies, their value cannot be changed once they are bound which means you cannot reassign them.

example:

```
const pi = 3.14;
```

Once defined with a type, a variable cannot change its type in TypeScript, it is therefore considered strongly typed.

In TypeScript, if a variable is not explicitly assigned a type when it is defined, then the type is inferred from the first assignment or the initialization of the variable and then it is then considered a strongly typed variable with the inferred type.

Types are classified into two main classes in TypeScript

1. Basic or Primitive Types: boolean, number, string, array, tuple, enum, null, undefined, any, void(exists purely to indicate the absence of a value, such as in a function with no return value)

2. Complex or Non-Primitive Types: class, interface

```
function f(){
    let someVariable = 1;
}

//If using let, someVariable will not be usable here.
//But when we use var, someVarible is usable here
```

You can use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by backtick/backquote(‘) character, and embedded expressions are of the from

```
${expr}
```

example

```
let fullName: string = ‘Bob Bobbington‘;
let age: number = 37;
let sentence: string = ‘Hello, my name is ${ fullName }.
I’ll be ${ age + 1 } years old next month.‘;
```

This is equivalent to declaring sentence like so:

```
let fullName: string = ‘Bob Bobbington‘;
let age: number = 37;
let sentence: string = "Hello, my name is " + fullName +
".\n\n" + "I’ll be " + (age + 1) + " years old next month.";
```

```
let isDone: boolean = false;
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let coler: string = "blue";
```

```
// TypeScript allows you to work with arrays of values.
// Array types can be written in one of two ways.
// In the first, you use the type of the elements followed by []
// to denote an array of that element type
let list1: number[] = [1, 2, 3];
// The second way uses a generic array type, Array<elemType>
let list2: Array<number> = [1, 2, 3];
```

Tuple: tuple allows you to express an array where the type of a fixd number of elements is known, but need not be the same. For example you may want to represent a pair of a string and a number:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10];
//Initialize it incorrrently
// x = [10, "hello"]
```

enum: A helpful addition to the standard set of datatypes from JavaScript is the enum. An enum is a way of giving more friendly names to sets of numeric values.

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

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0.

```
enum Color {Red = 1, Green, Blue};
let c: Color = Color.Green;
```

Or manually set all values in the enum.

```
enum Color {Red = 1, Green = 2, Blue = 4};
let c: Color = Color.Green;
```

A handy feature of enums if that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren't sure what that mapped to in the Color enum above, we could look up the corresponding name:

```
enum Color { Red = 1, Green = 2, Blue = 5 };
let str: string = Color[5];
alert(str);
```

any

```
let notSure: any = 4;
notSure = "maybe a string instead"
notSure = false;
```

void: void type is mainly used as the return type of functions that do not return a value.

```
function warnUser(): void{
    alert("This is my warning message");
}
```

Declaring variables of type void is not useful because you can only assign undefined or null to them.

```
let unusable: void = undefined
unusable = null;
```

null and undefined: By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

Avoid using any

```
let x;// let x: any;
x = 3;
x = "1";
```

Type assertions: type assertions are like type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact and is used purely by the compiler.

Type assertions have two forms. One is the "angule-bracket" syntax.

```
let someValue: any = "this is a string";
let strLength: number = (<string> someValue).length;
```

And the other is the as-syntax.

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

4 Complex types

4.1 Interface

If an instance of an object satisfies the shape of an interface, then the object can be assigned to any variable defined as that interface type.

Example:

```
function printLabel(labelledObj:{label: string}){
    console.log(labelledObj.label)
}
let myObj = {size: 10, label:"Size 10 object"}
printLabel(myObj)
```

The printLabel function has a single parameter that requires that the object passes in has a property called label of type string. Notice that our object actually has more properties than this, but the compiler only checks that at least the ones required are present and match the types required.

We can write the same example again.

```
interface LabelledValue{
    label: string;
}
function printLabel(labelledObj:LabelledValue){
    console.log(labelledObj.label)
}
let myObj = {size: 10, label:"Size 10 object"}
printLabel(myObj)
```

Not all properties of an interface may be required.

```
interface SquareConfig{
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): {color: string; area:
    number}{
    let newSquare = {color: "white", area: 100};
    if(config.color){
        newSquare.color = config.color;
    }
    if(config.width){
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

let mySquare = createSquare({color: "black"});
```

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a ? at the end of the property name in the declaration.

readonly properties: Some properties should only be modifiable when an object is first created. You can specify this by putting readonly before the name of the property:

```
interface Point{
    readonly x: number;
    readonly y: number;
}
let p1: Point ={x:10, y:20};
// p1.x = 5; // error!
```

$ReadonlyArray < T >$ type is the same as $Array < T >$ with all mutating methods removed, so you can make sure you don't change your arrays after creation:

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
//Error!
//ro[0] = 12;
//ro.push(5);
//ro.length = 100;
//a = ro;
```

Even assigning the entire ReadonlyArray back to a normal array is illegal. You can still override it with a type assertion, though:

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
a = <number[]>ro;
a = ro as number[];
```

4.2 class types

Implementing an interface:

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date){
        this.currentTime = d;
    }
    constructor(h: number, m: number){}
}
```

Interface describes the public side of the class, rather than bothe public and private side. This prohibits you from using them to check that a class also has particular types fot the private side of the class interface.

5 SOLID

Single Resonsibility Principle, Open/Closed Principle, Liskov substitution Principle, Interface Segregation Principle, Dependency Inversion Principle

6 Writing a simple markdown parser

Our starting point is this page, which strtches across the full width of the screen by setting the container to use container-fluid, and divides the interface into two equal parts by setting col-lg-6 on both sides.

We need to manually set the style of the html and body tags to fill the available space.

We apply h-100 to these classes to fill 100% height of the space.

6.1 Visitor pattern

What the visitor pattern gives us is the ability to separate an algorithm from the object that the algorithm works on.

6.2 Reminders

1. document.getElementById

2. value attribute of HTMLTextAreaElement is the content of the input.

3. startsWith function of string

4. substr function of string: Two parameters, first parameter is the start index of the substring, and the second parameter (optional) if the length of the substring.

5. innerHTML attribute of HTMLTextAreaElement.

6. length attribute of string

7. Use CSS class ”container-fluid” for a full width container, spanning the entire width of your viewport.

```
<div class="container-fluid">
</div>
```

8. Use CSS class ”row” to create a row.

9. Make body 100% height of the page.

```
<style>
  html, body {
    height: 100%;
  }
</style>
```