Week 7.2 Intro to typescript.

Why Typescript:-

1. Type script is a superset of JS.
2. It gives you static Typing.
3. Makes your code more strict. Hence avoids runtime errors.

Lets convert a small JS file to TS.

JS.

```
let x = 10;
x = "Hello";
```

TS.

```
let x: number = 10;
x = "Hello";
```

Code IDE will give error as you are reassigning a variable of which type is number but you are now giving it a string.

★ This is what we do in case of functions.

```
1  // TS
2  function calculateSum(first: number, second: number) {
3    return first + second;
4  }
5
6  let answer = calculateSum(1, 2);
7  console.log(answer);
```

returning number as well.

* What we need now is, what if the arguments are not just number / str but the arguments only can be add, div, mul, sub? How will we deal with this case?

→

```ts
function calculate(a: number, b: number, type: string): number {
  if (type === "sum") {
    return a + b;
  }
  if (type === "sub") {
    return a - b;
  }
  if (type === "mul") {
    return a * b;
  }
  if (type === "div") {
    return a / b;
  }
} return -1;
  edge case.

const ans = calculate(1, 2, "sum");
console.log(ans);
```

also we are not returning the edge case here!

This solution is good if the user only gives sum, sub, mul & div as type, but what if the user gives xyz or abc as input the the function will return undefine which is not good for an enterprise app.

SO The above soln is not good.

Problem ?

It's not strict enough

```ts
// TS
function calculateSum(
  first: number,
  second: number,
  type: string
) {
  if (type === "div") {
    return first / second;
  }
  if (type === "mul") {
    return first * second;
  }
  if (type === "sub") {
    return first - second;
  }
  if (type === "add") {
    return first + second
  }
  return -1;
}

let answer = calculateSum(1, 2, "randomGiberish");
console.log(answer);
```

This is the issue

Solution

Union operator

```ts
function calculateSum(
  first: number,
  second: number,
  type: "add" | "sub" | "mul" | "div"
) {
  if (type === "div") {
    return first / second;
  }
  if (type === "mul") {
    return first * second;
  }
  if (type === "sub") {
    return first - second;
  }
  if (type === "add") {
    return first + second
  }
  return -1;
}

let answer = calculateSum(1, 2, "randomGiberish");
console.log(answer);
```

This is "or" '|'

By this ts will not allow any input for type other than add, sub, mul or div

☆ Typescript never runs our code it just converts it into js. Bas.

TS vs JS

How does TS run code? It never does

It does

1. Type checking
2. converts to JS

```ts
function calculateSum(
  first: number,
  second: number,
  type: "add" | "sub" | "mul" | "div"
) {
  if (type === "div") {
    return first / second;
  }
  if (type === "mul") {
    return first * second;
  }
  if (type === "sub") {
    return first - second;
  }
  if (type === "add") {
    return first + second
  }
  return -1;
}

let answer = calculateSum(1, 2, "add");
console.log(answer);
```

→

```js
"use strict";
// TS
function calculateSum(first, second, type) {
    if (type === "div") {
        return first / second;
    }
    if (type === "mul") {
        return first * second;
    }
    if (type === "sub") {
        return first - second;
    }
    if (type === "add") {
        return first + second;
    }
    return -1;
}
let answer = calculateSum(1, 2, "add");
console.log(answer);
```

What is the TS compiler?

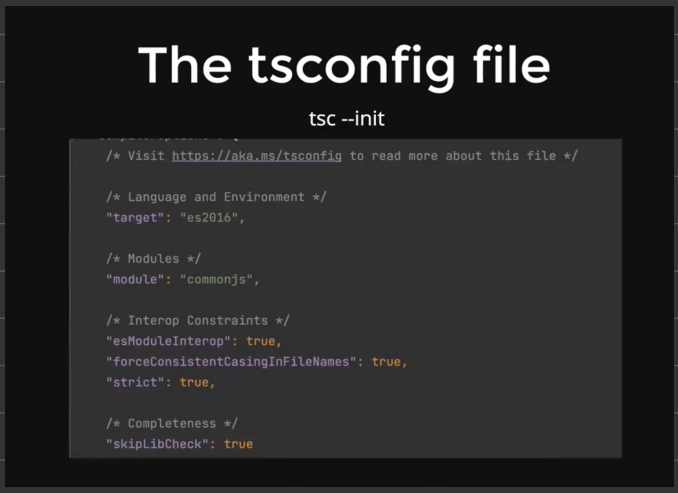
☆ The compiler is called tsc.

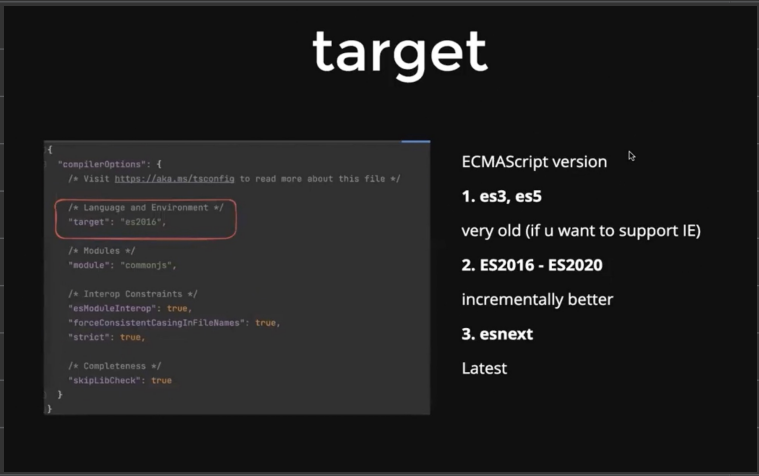
☆ To bring tsc on you machine locally you need to do " npm i -g tsc" & install it as a global dependency / Package.

☆ To convert your .ts file to a .js file you need to do " tsc sum.ts" & "tsc" and in the end it will create a "sum.js" file just beside the sum.ts file.

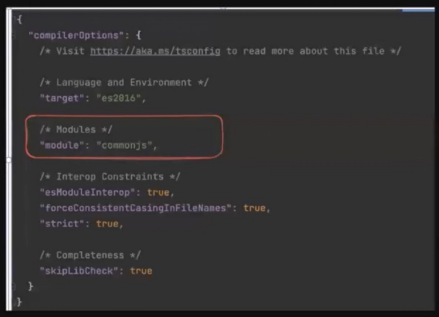
The tsconfig file.

☆ To create a ts config file in your directory. do " tsc --init ".





module



module type - target system

1. none

This isn't a module

2. commonjs

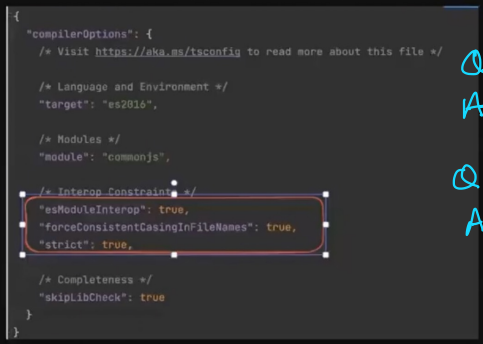
Node.js code

3. amd

browser code

Lets try commonjs vs es6

Constraints



Q · forceConsistentCasingInFileNames

A → Does casing matter when importing modules

Q strict

A · How strictly should TS be evaluated

Interfaces

(Person interface)

1. Let you accumulate data of a specific type
2. Interfaces can use other interfaces
3. Interfaces can extend interfaces
4. Interfaces can be implemented by classes

```
export function greet(person: {
    name: string;
    age: number
}): string {
    return "Hello mr " + person.name + " glad that you are " + person.age  + "years old
}

console.log(greet({
    name: "harkirat",
    age: 21
}))
```

This is also a way to pass complex data types in ts but it is the ugly way. Instead of using this types of syntax we can use INTERFACES.

```
// string, number
interface Person {
    name: string;
    age: number;
}

export function greet(person: Person): string {
    return "Hello mr " + person.name + " glad that you are " + person.age  + "years old
}

export function agePrint(person: Person): string {
    return "You are " + person.age  + "years old";
}

console.log(greet({
    name: "harkirat",
    age: 21
}))
```

* Interfaces must be named with first letter capital.

* This is how we must deal with complex data types like objects in TS.

Here is the Person interface

```ts
interface PersonInterface {
    name: string;
    age: number;
    greet(): string;
}

class Person implements PersonInterface {
    name: string;
    age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet() {
        return "hi mr " + this.name;
    }
}

const personObject = new Person("harkirat", 21)
console.log(personObject.greet())
```

In JS it is not needed But in ts it is imp to predeclare properties in classes.

* Interfaces can be implemented in classes. & a class can implement multiple interfaces. using the implements keyword.
* One thing to keep in mind is which ever interface is implemented by a class in ts, It must have all the properties used inside of the class & its constructor e.g like you can see in the Person class it hass name & age same as PersonInterface.

```ts
interface PersonInterface {
    name: string;
    age: number;
}

function greet(person: PersonInterface) {

}

console.log(greet({
    name: "harkirat",
    age: 21
}))
```

forget what we did in the above example but this is the use-case which you use in 99% of the cases.

Interfaces can use other interfaces in their declarations.

```ts
interface PersonGenderProperties {
    gender: string;
    orientation: string;
    pronouns: string;
}

interface PersonInterface {
    name: string;
    age: number;
    genderProps: PersonGenderProperties;
}

function greet(person: PersonInterface) {

}

console.log(greet({
    name: "harkirat",
    age: 21,
    genderProps : {
        gender: "male",
        orientation: "straight",
        pronouns: "he/him"
    }
}))
```

Here we used the PersonGenderProperties inside the PersonInterface.

usage

```ts
interface PersonGenderProperties {
    gender: string;
    orientation: string;
    pronouns: string;
}

interface PersonInterface extends PersonGenderProperties{
    name: string;
    age: number;
}

interface AnimalInterface extends PersonGenderProperties {
    name: string;
    furType: string;
}
```

* Interfaces can extend other interfaces using the extends keyword.

Now this PersonInterface will also have the properties gender, orientation & pronouns along with name & age.

Advantage of using the extends keyword the properties are mandatory of the base interface to the consumer without properly declaring each property.

TYPES in TS.

* Types are sort of same as interfaces but come with some limitations. as below.

1. Types cannot use extends keyword & hence not extend properties of other types or interfaces.
2. Types cannot use implements keyword & hence cannot be used in classes to implement their properties.

— other things are same

```ts
type PersonGenderProps = {
    gender: string;
}

type PersonInterface = {
    name: string;
    age: number;
    gender: PersonGenderProps;
}

function greet(person: PersonInterface) {
    return "Hi mr " + person.name + " your age is " + person.age;
}

console.log(greet({
    name: "harkirat",
    age: 21,
    gender: {
        gender: "nale"
    }
}))
```

Types are very useful for unions and ors.

```ts
interface Circle {
    radius: number;
}

interface Square {
    side: number;
}

interface Rectangle {
    width: number;
    height: number;
}

type Shape = Rectangle | Circle | Square;

function renderShape(shape: Shape) {
    console.log("Rendered!");
}

function calculateArea(shape: Shape) {
    console.log("cacluated area");
}
renderShape({
    radius: 10
})
```

This is OR

for any one property on shape form Rect, Circle, Square.

any one property

```ts
interface Circle {
    radius: number;
}

interface Square {
    side: number;
}

interface Rectangle {
    width: number;
    height: number;
}

type Shape = Rectangle & Circle & Square;

function renderShape(shape: Shape) {
    console.log("Rendered!");
}

function calculateArea(shape: Shape) {
    console.log("cacluated area");
}
renderShape({
    radius: 10,
    width: 10,
    height: 10,
    side: 10
})
```

This is union

Shape will now require all of the properties from Rectangle & Circle & Square

needs all of the properties.

- Interfaces not allow unnions & ors like types
- If you have unions in types & any two types of the union have any one same property then the final union type will require you to specify that particular property only once, irrespective of how many types from the union chain "A & B & C"

There's one imp point to remmember if the common property has same key but different value type like string for one & number for other then you must give both properties.

ex

```ts
interface Circle {
    radius: number;
}

interface Square {
    side: number;
}

interface Rectangle {
    side: number;
    height: number;
}

type Shape = Rectangle | Circle | Square;

function renderShape(shape: Shape) {
    console.log("Rendered!");
}

function calculateArea(shape: Shape) {
    console.log("cacluated area");
}
renderShape({
    radius: 10,
    width: 10,
    height: 10,
    side: 10
})
```

1st

2nd

1st 2nd

only one here.

Ln 15

TS sum.ts > ...

```ts
interface Circle {
    radius: number;
    borderWidth?: number;
}
```

optional key/property while using. if you need optional values in interfaces.

Enums

TS sum.ts > x

```ts
type Arithmentic = "add" | "sub" | "div" | "mul";

function calculateSum(a: number,
    b: number,
    type: Arithmentic) {
    // body
}

let x = calculateSum(1, 2, "div");
console.log(x);
```

TS sum.ts > ...

```ts
enum Arithmentic {
    Add,
    Sub,
    Div,
    Mul
};

function calculateSum(a: number,
    b: number,
    type: Arithmentic) {
    // body
}

let x = calculateSum(1, 2, Arithmentic.Add);
console.log(x);
```

This makes the unwanted hassel of what if the user gives "division" instead of 'div'

TS sum.ts > Shape

```ts
enum Arithmetic {
    Add,
    Sub,
    Div,
    Mul
}

enum Shape {
    Circle,
    Square,
    Rect
}

enum CourseType {
    Paid,
    Free
}

function calculate(a: number, b: number, type: Arithmetic) {
    ///
    return 1;
}

const ans = calculate(1, 2, Arithmetic.Div)
```

```ts
enum Arithmetic {
    Add, = 0
    Sub, = 1
    Div, = 2
    Mul  = 3
}


function calculate(a: number, b: number, type: Arithmetic) {
    console.log(type);
    /// div  -> 1
    /// div
    /// Arithmetic.Div
    /// 0
    /// 1
    /// 2
    return 1;
}

const ans = calculate(1, 2, Arithmetic.Div)
```

Ans

Because enums assigns an id of 0,1,2,3 to all its keys like
∴ ans is (2.)

This is how tsc compiles enums.

JS sum.js > ...

```
1   "use strict";
2   var Arithmetic;
3   (function (Arithmetic) {
4       Arithmetic[Arithmetic["Add"] = 0] = "Add";
5       Arithmetic[Arithmetic["Sub"] = 1] = "Sub";
6       Arithmetic[Arithmetic["Div"] = 2] = "Div";
7       Arithmetic[Arithmetic["Mul"] = 3] = "Mul";
8   })(Arithmetic || (Arithmetic = {}));
9   function calculate(a, b, type) {
10      console.log(type);
11      return 1;
12  }
13  const ans = calculate(1, 2, Arithmetic.Div);
```

☆ You can return enum.key but not interfaces & types.