One Must Read Documentation to set the things done

Types:

Number	String	Boolean	
Null	Undefined	Void	
Object	Array	Tuples	& so on ....

If you using Any keyword then it more of type of Javascript code

Situations

- A function accepts 2 numbers
- A function returns a string

Situations where we can use TypeScript

① Fn accept 2 no's Now in JS there can be any type of parameters of fn so need to check But here we have types

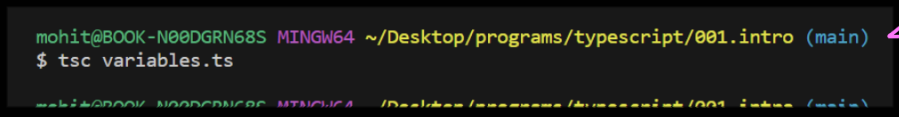
② Fn returns a string in JS we can't say what a fn will return but here we are sure what a fn will return.

```
let variableName: type = value
```

let / const / var whatever

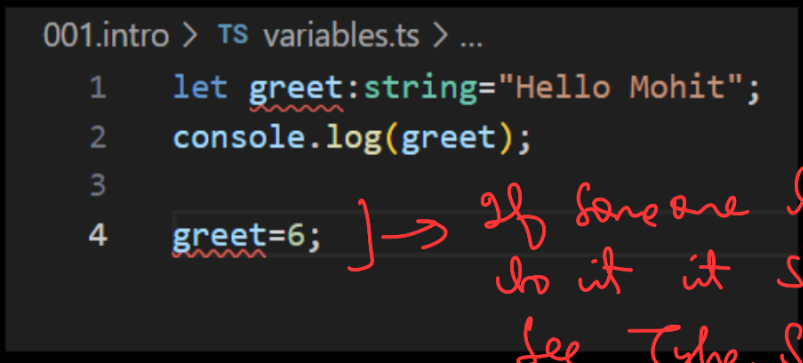
All type in TypeScript is in Lower case

tells greet is a string

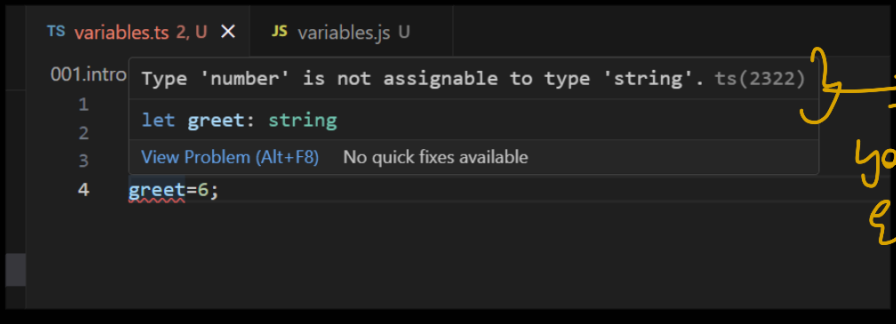


generates .js file

Will see error on line 1 later



If someone later tries to assign to it it shows Red line see Type Safety !!



See on hovering you even get the error

The primitives: `string`, `number`, and `boolean`

JavaScript has three very commonly used primitives: `string`, `number`, and `boolean`. Each has a corresponding type in TypeScript. As you might expect, these are the same names you'd see if you used the JavaScript `typeof` operator on a value of those types:

- `string` represents string values like `"Hello, world"`
- `number` is for numbers like `42`. JavaScript does not have a special runtime value for integers, so there's no equivalent to `int` or `float` - everything is simply `number`
- `boolean` is for the two values `true` and `false`

001.intro > TS variables.ts > ...

```ts
let greet:string="Hello Mohit";
console.log(greet);

greet=6;

//number

let i:number=554441;

let u:number=123.3211;

let check:boolean=true;
```

Rest of two type Usage

Here its faltu Mai likha hue hai Number its obvious i is a Number as we are immediately assigning value to it. Type Script is Smart Enough to decide type from the assigned Value.

Not Always Every Variable should have declared with Type. Type Script is Smart Enough to get Type from the assigned value. This process is called Type Inference.

TS variables.ts 4, U | JS variables.js U

001.intro > TS variables.ts > ...

```ts
let greet:string="Hello Mohit";
console.log(greet);



//number

let i:number=554441;

let u:number=123.3211;

let check:boolean=true;
```

After Compiling →

TS variables.ts 4, U | JS variables.js U

001.intro > JS variables.js > ...

```js
var greet = "Hello Mohit";
console.log(greet);
//number
var i = 554441;
var u = 123.3211;
var check = true;
```

Any Keyword

Here hero is any which means type of hero is not defined. Any is given to type when not given any type

any

TypeScript also has a special type, `any`, that you can use whenever you don't want a particular value to cause typechecking errors.

no typechecking is done therein Any

When a value is of type `any`, you can access any properties of it (which will in turn be of type `any`), call it like a function, assign it to (or from) a value of any type, or pretty much anything else that's syntactically legal:

```
let obj: any = { x: 0 };
// None of the following lines of code will throw compiler errors.
// Using `any` disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.foo();
obj();
obj.bar = 100;
obj = "hello";
const n: number = obj;
```

The `any` type is useful when you don't want to write out a long type just to convince TypeScript that a particular line of code is okay.

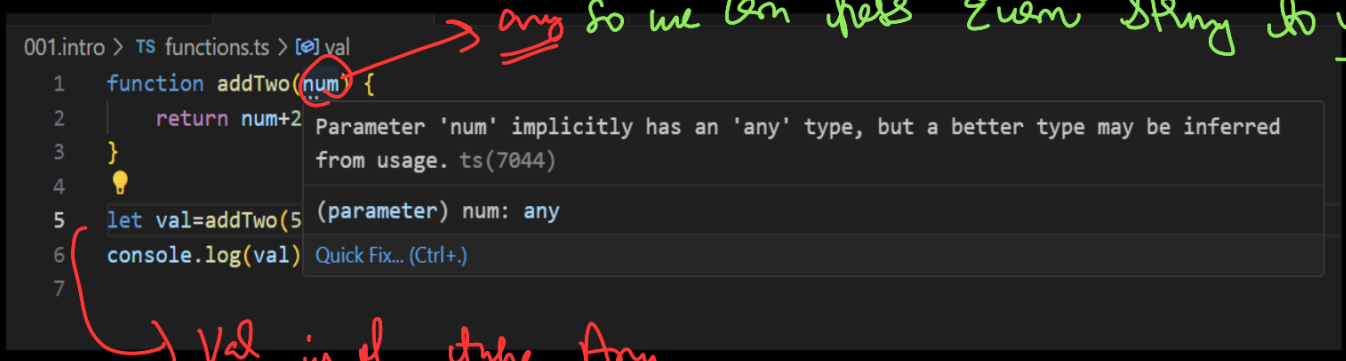
noImplicitAny

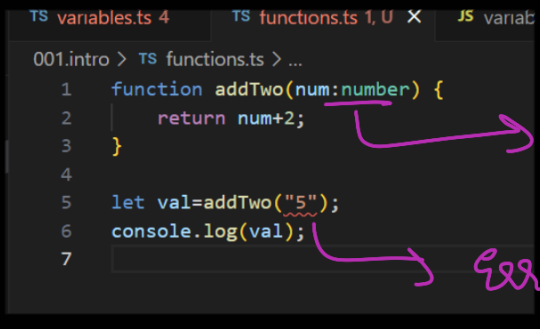
When you don't specify a type, and TypeScript can't infer it from context, the compiler will typically default to `any`.

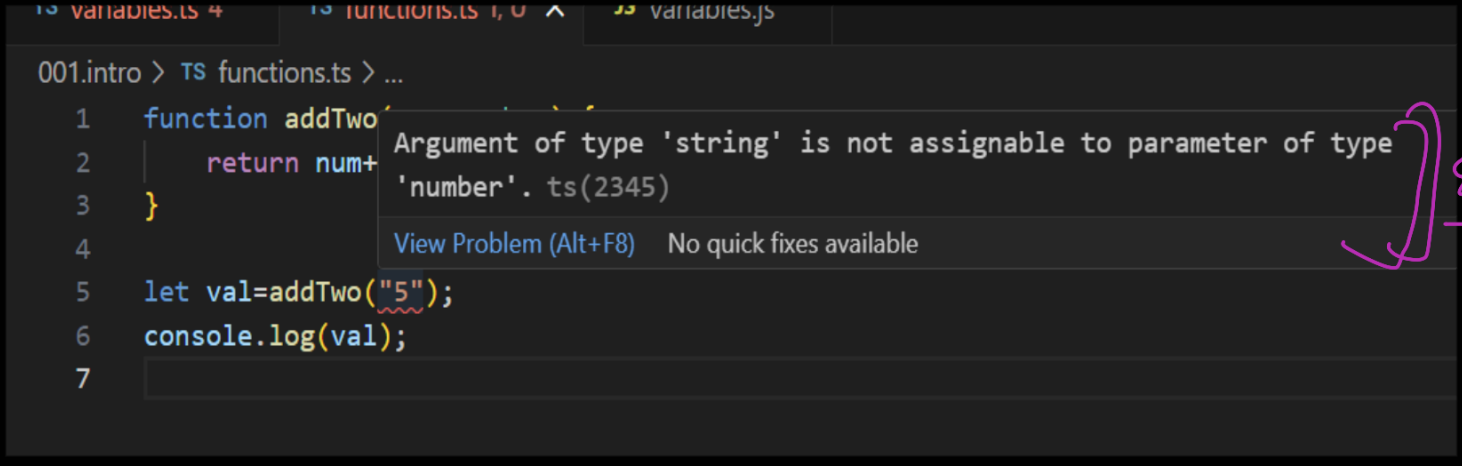
You usually want to avoid this, though, because `any` isn't type-checked. Use the compiler flag `noImplicitAny` to flag any implicit `any` as an error.

Using Any is not a good practice. With Any you are moving towards JS type code & not of TypeScript Type code

functions







For fn parameters we always declare a type with it

TypeScript doesn't show all Error One. It shows one Error One Once you correct a Error then next Error is shown

Arrow fn

```
let user=(name:string,password:string,isADMIN:boolean=false)=>{
    if(!isADMIN){
        console.log("not admin "+name);

    }
}

user("mohit","1245",true);
user("rohit","1245");
```

Default Value is false

Both are Valid

TypeScript help us to write cleaner code

```
let user=(name:string,password:string,isADMIN:boolean=false)=>{
    if(!isADMIN){
        console.log("not admin "+name);
        return "hello";

    }
    return false;

}

console.log(user("mohit","1245",true));
console.log(user("rohit","1245"));
```

See here we are not going to handle the type of return type once it returns "hello" & other return is false

TypeScript is just a way to write better JS.

```
let user=(name:string,password:string,isADMIN:boolean=false):number=>{
    if(!isADMIN){
        console.log("not admin "+name);
        return 1;

    }
    return 2;

}

console.log(user("mohit","1245",true));
console.log(user("rohit","1245"));
```

Here we provide return type

Sometimes we want different return type so for that we use Union type later

```
31 const heros = ["thor", "spiderman", "ironman"]
32 // const heros = [1, 2, 3]
33            (parameter) hero: string
34 heros.map(hero => {
35     return `hero is ${hero}`
36 })
37
```

```
31 //const heros = ["thor", "spiderman", "ironman"]
32 const heros = [1, 2, 3]
33            (parameter) hero: number
34 heros.map(hero => {
35     return `hero is ${hero}`
36 })
37
```

Typescript Automatically detects which type of content it is passed to

```
21
22  let arr=["a","b","c"];
23
24  arr.map((ele):string=>{
25      return "Elment is "+ele;
26  })
```

Here see we have Error but what return value is Expected from the map callback

Void fn
So void is close Type

```
38
39 function consoleError(errmsg: string): void{
40     console.log(errmsg);
41
42 }
```

fn never returns a value when we want to throw an Error

```
43 function handleError(errmsg: string): never{
44     throw new Error(errmsg);
45
46 }
47
```

The `never` type represents values which are *never* observed. In a return type, this means that the function throws an exception or terminates execution of the program.

`never` also appears when TypeScript determines there's nothing left in a union.

Some bad behaviour in TypeScript at Objects

```ts
// 002.basics > TS Objects.ts > ...
const User = {
    name: "hitesh",
    email: "hitesh@lco.dev",
    isAvtive: true
}

function createUser({name: string, isPaid: boolean}){}

```

Here we are telling ki fn createuser ke parameter mei object jisme name string hoga & isPaid boolean hoga

Wierd part here ??
we are passing Email although fn doesn't Expect that !! Yes working perfectly

To access value with Type check

```ts
// 002.basics > TS Objects.ts > createUser
const User = {
    name: "hitesh",
    email: "hitesh@lco.dev",
    isAvtive: true
}

function createUser({ name, isPaid }: { name: string, isPaid: boolean }) {
    console.log("Name is"+name);
    console.log("Ispaid value"+isPaid);

}

let newUser = {name: "hitesh", isPaid: false, email: "h@h.com"}

createUser(newUser)

```

Now lets see fn returning Objects !!

function Keyword fn Name (parameters) : { } { }
return type fn body

```ts

function createCourse():{name: string, price: number}{
    return {name: "reactjs", price: 399}
}

```

fn returning objects

Weirdness of Objects

```
function createUser({ name, isPaid }: { name: string, isPaid: boolean }) {
    console.log("Name is"+name);
    console.log("Ispaid value"+isPaid);

}

let newUser = {name:"mohit", isPaid: false, email: "h@h.com"}

createUser(newUser)
```

not passed as parameter but still works

we should not be able to pass email

```
function createUser({ name, isPaid }: { name: string, isPaid: boolean }) {
    console.log("Name is"+name);
    console.log("Ispaid value"+isPaid);

}

//let newUser =

createUser({name:"mohit", isPaid: false, email: "h@h.com"})
```

Yaaa, it shows error why ??

Type Aliases

```
type User = {
    name: string;
    email: string;
    isActive: boolean
}


function createUser(user: User): User{
    return {name: "", email: "", isActive: true}
}
```

Here user is Type Alias !!

Now we can directly pass User & not the object need to be specified again & again !!

A *type alias* is exactly that - a *name* for any *type*. The syntax for a type alias is:

```
type Point = {
  x: number;
  y: number;
};

// Exactly the same as the earlier example
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

} like a class

Suppose we have 16 attributes inside Object & 10 fn having usage of that so there we can use these