Typescript Intro

Types of languages

1. **Strongly typed vs loosely typed**

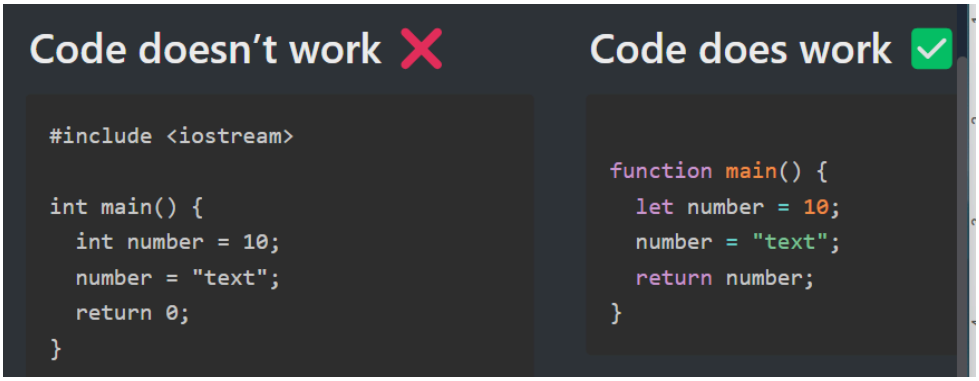
The terms strongly typed and loosely typed refer to how programming languages Handle types, particularly how strictly they are about type conversions and type safety.

Strongly typed languages

1. Java, C++, C, Rust
2. Benefits:
    - Lesser runtime errors
    - Stricter codebase
    - Easy to catch errors at compile time

Loosely typed languages

1. Python, Javascript
2. Benefits:
    - Easy to write code
    - Fast to bootstrap
    - Low learning curve



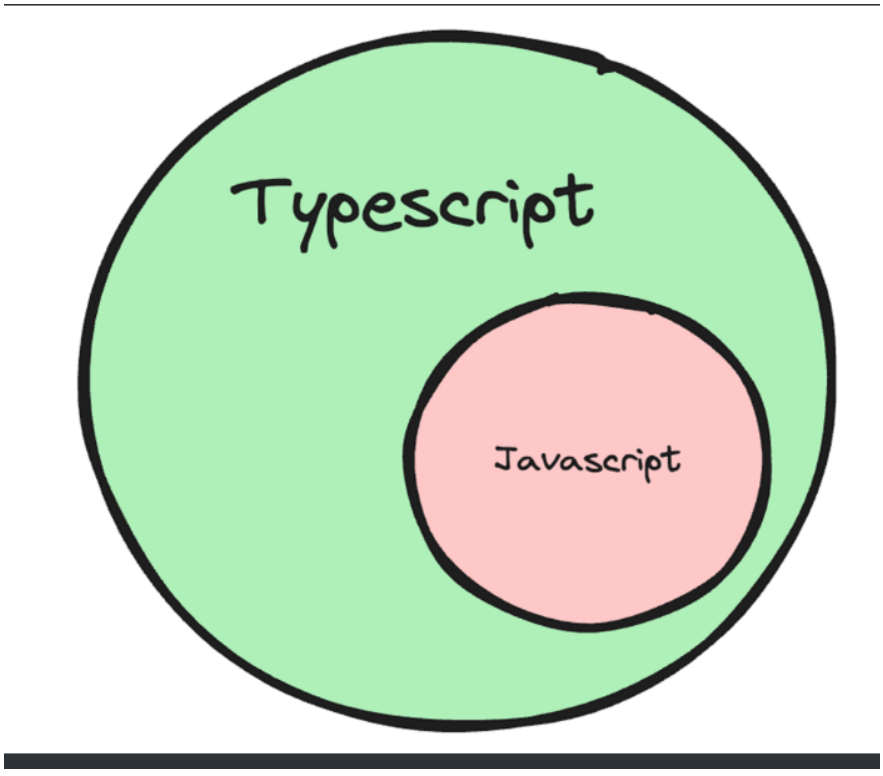
Typescript was introduced as a new language to add types on top of javascript

Strongly typed language . Error are caught at compile time (typed)
In strongly typed language there are less run time error because we catch errors during compile time.

By the end if it compiles

What is TypeScript

Typescript is a programming language developed and maintained by Microsoft.
It is a strict **syntactical** superset of Javascript and adds optional static typing to the languager.

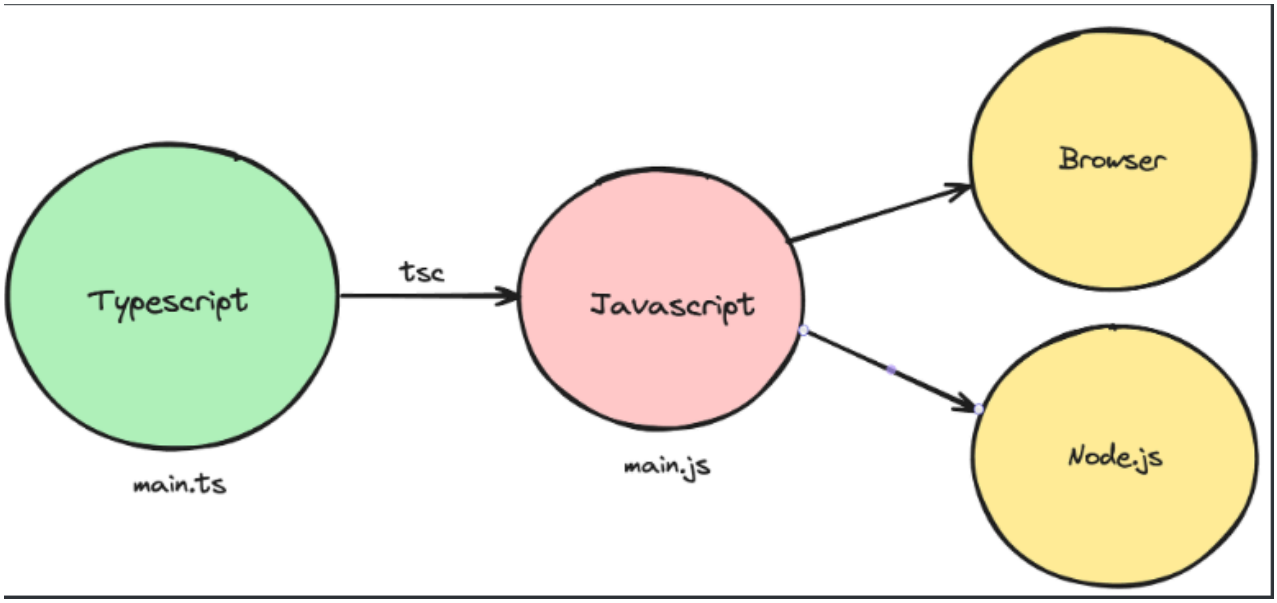


Where/ How does typescript works.

Typescript is backward compatible.

Typescript code never runs in your browser. Your browser can only understand **javascript.**

1. Js is the runtime language (the thing that actually runs in your browser/nodejs runtime)
2. Typescript is something that compiles(transpile) down to Javascirpt.
3. When typescript is compiled down to Js ,you get **type checking** (Similar to C++). If there is an error the conversion to Javascript fails.



We can run command to convert ts into js
Typescript compiler will do all the static type check and then if all valid then only gives Js file.
Typescript compiler ensures that there are no type error in typescript file.

Typescript compiler

Tsc is the official typescript compiler that you can use to convert **typescript** code into the Javascript
There are many other famous compilers/transpilers for converting Typescript to Javascript some famous are –

1. esbuild
2. swc

The tsc compiler

Lets bootstrap a simple Typescipt Node.js application locally on your machine.

Step1 - Install tsc/ typescript globally

npm install -g typescript

Step 2 - Initialize an empty Node.js project with typescript

mkdir node-app

cd node-app

npm init -y

npx tsc --init

tsconfig.json

This file consist of bunch of configuration/variable based on which Our compilation process changes.

How do we want to convert our transcript file into a javascript one is contained inside it.

package.json has to do nothing with typescript

a.ts

```
const x:number = 1;
// similar to what we write in c++ int x = 1
console.log(x);
```

tsc

tsc -b

(Compile the ts file to js file)

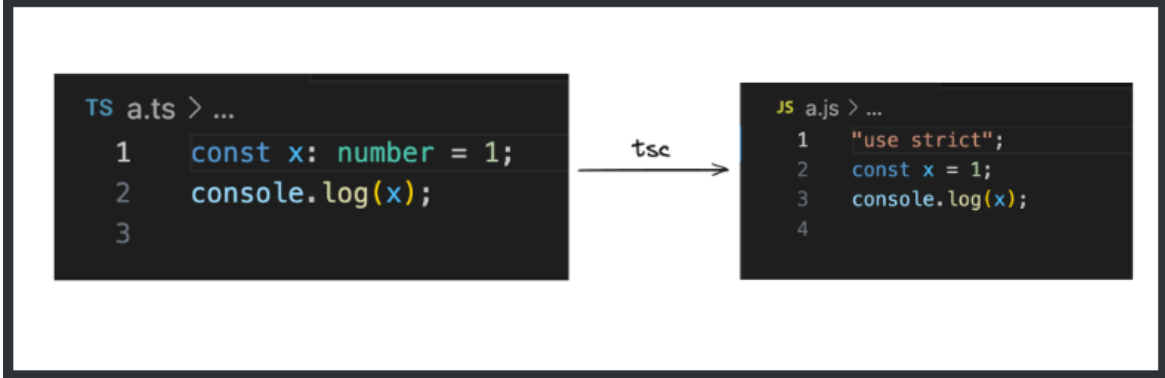
File typescript compiler generated

a.js

```
"use strict";
const x = 1;
```

```
// similar to what we write in c++ int x = 1
console.log(x);
```

Step 5 - Explore the newly generated index.js file



Notice how there is no typescript code in the javascript file. It's a plain old js file with no `types`

To run our code we will do **node a.js**

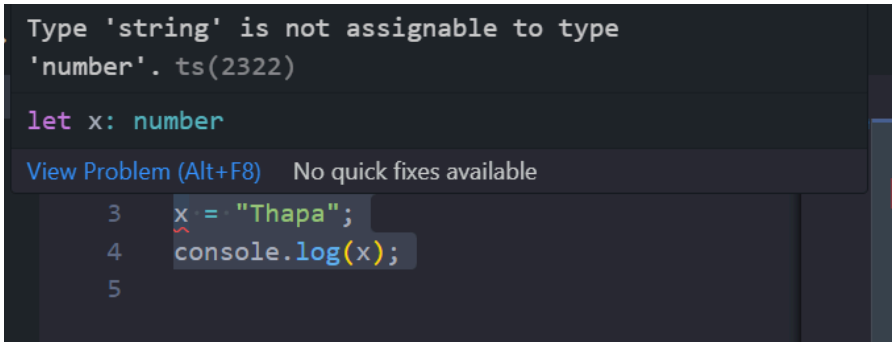
Node doesn't understand a.tx file.

Now delete a.js

And introduce something which is wrong(in typescript file**).**

```
let x:number = 1;
// similar to what we write in c++ int x = 1
x = "Thapa";
console.log(x);
```

It shows error



Typescript add typesafety to our Javascript code. In most codebases hence typescript is used.

Basic Types in Typescript

Typescript provides some basic types

number, string, boolean, null, undefined

Let's create some simple application using these types -

Problem 1- Hello world

Write a function that greets a user given their first name.

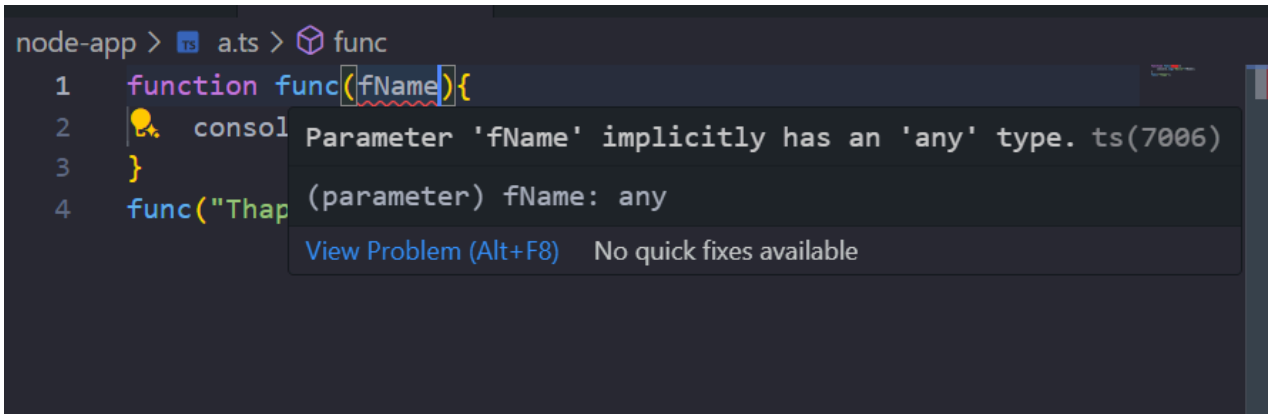
Argument - firstName

Logs - Hello {firstName}

Doesn't return anything

Solution:

```
function func(fName: string){
    console.log("Hello"+fName);
}
func("Thapa");
```

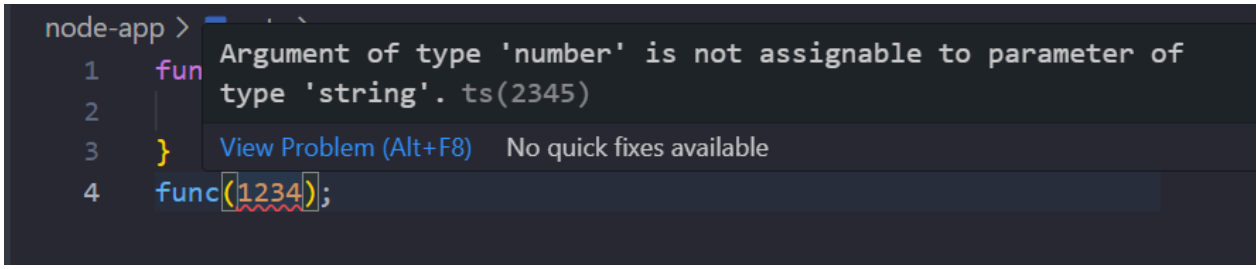


Just like string, number any is also a type.

We can explicitly tell them any then it will work

let x: any = 1;

Let's try to give a wrong input.



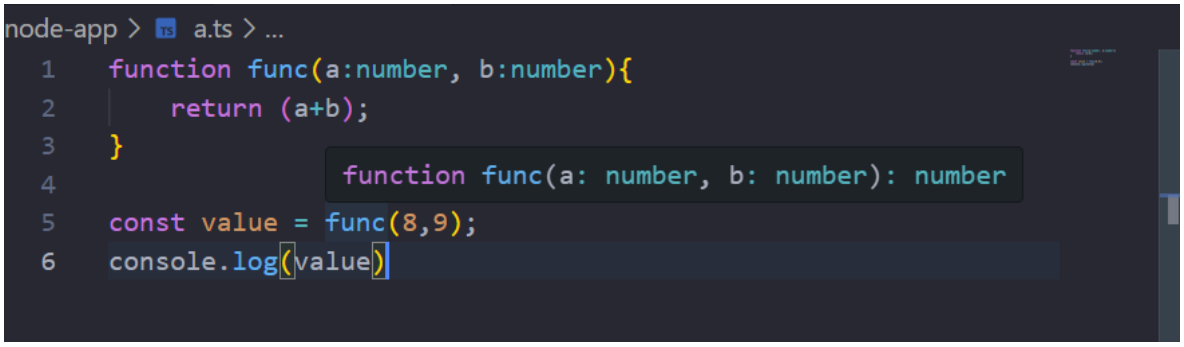
Now suppose we are in other component and see this function , just hovering it we can see what type of input it takes.

Problem 2 – Sum function

Write a function that calculates the sum of two functions

```
function func(a:number, b:number){
    return (a+b);
}

const value = func(8,9);
console.log(value)
```



Here we can see typescript figured out what will the function return

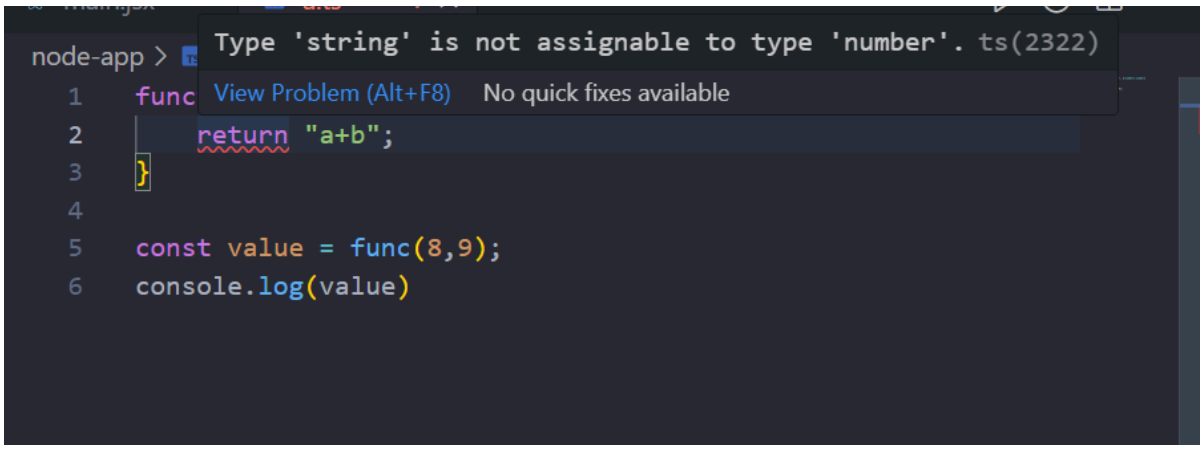
Typescript was able to infer the return type.

It is good practice to explicitly define the type

Lets See how to explicitly define what to return

```
function func(a:number, b:number):number{
    return (a+b);
}

const value = func(8,9);
console.log(value)
```



Typescript doesn;t verify code, it only verifies type.

Problem 3 – Return true or false based on if a user is 18+

```
function isLegal(age: number) {
    if (age > 18) {
        return true;
```

```
    } else {
        return false
    }
}

console.log(isLegal(2));
```

Implicitly typescript able to refer the type

Problem 4 –

Create a function which take another function as input and runs it after 1 seconds.

```
function runAfter1s(fn: () => void){
    setTimeout(fn,1000);
}

runAfter1s(function(){
    console.log("hi there")
    // not returning anything
})
```

Here the thing to learn is how to give type to a function.

The tsconfig file

The tsconfig file has a bunch of options that you can change to change the compilation process.

Some of these include

1. **target**

The target option in a tsconfig.json file specifies the ECMAScript target version to which the TypeScript compiler will compile the TypeScript code.

Try following code in ES5 and es2020

const greet = {name:string} => `Hello, ${name}!`;

Output for ES5

```
"use strict";
var greet = function (name) { return "Hello, ".concat(name, "!"); };
```

Output for ES2020

```
"use strict";
const greet = (name) => `Hello, ${name}!`;
```

Arrow function was later introduced in the ECMA script

2. **rootDir**

Where should the compiler look for .ts files. Good practice is for this to be the **src** folder. It's very hard to account for changes when ts and js files are both in same folder . We create a source **src** folder and put all the typescript code here. All final output lie in **dist / build**

3. **outDir**

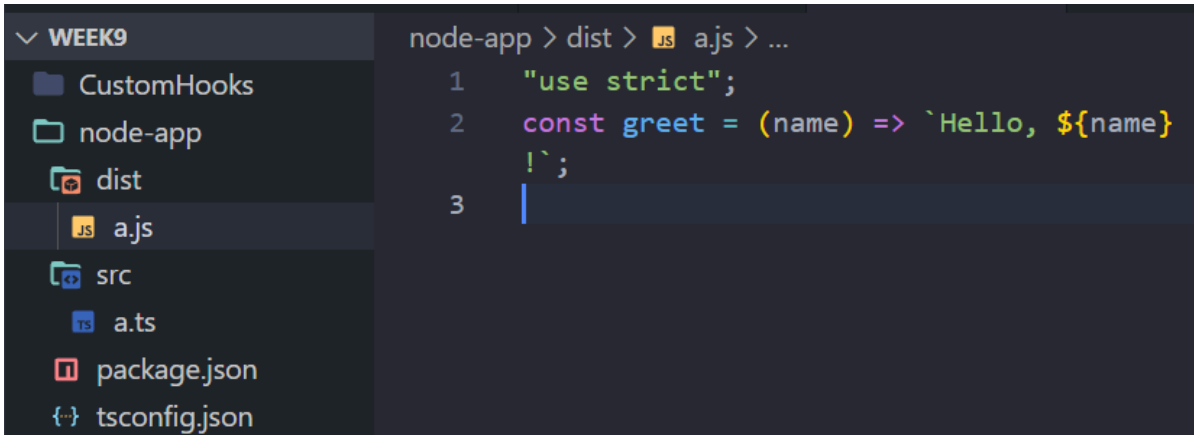
Where should the compiler look for spit out the **.js file.**

tsconfig.json

```
{
  "compilerOptions": {

    "target": "ES2016",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist",
    "esModuleInterop": true,
"forceConsistentCasingInFileNames": true,
    "strict": true,                                    /
    "skipLibCheck": true                                 }
}
```

Now we can see that a.js file is created inside dist folder which we don't touch



node dist/a.js will run the code

We never push the final file dist to the github. Only the original typescript code get committed to the github

We do tsc -b locally to get that dist folder containing the final js files

4. **noImplicitAny**

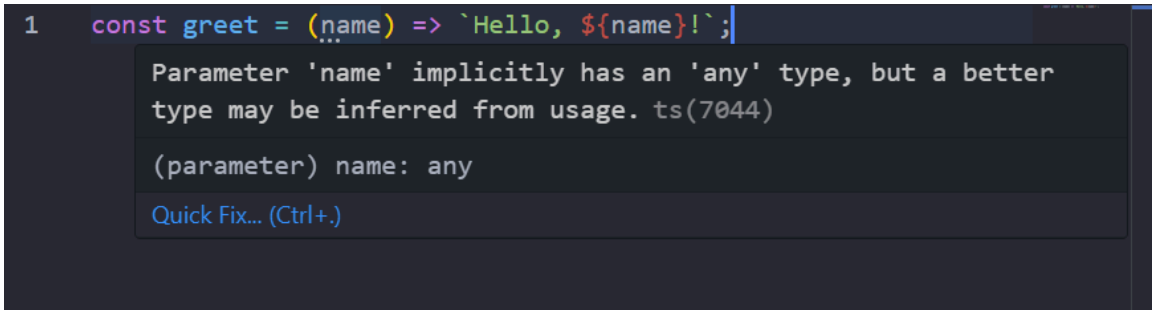
Try enabling it and see the compilation errors on the following code -

const greet = (name) => `Hello, ${name}!`;

Then try disabling it

"noImplicitAny":false – make our code base less strict, useful when we


```
"noImplicitAny": true,                              /* Enable error
reporting for expressions and declarations with an implied 'any' type. */
```



Now there is no error when we change

“noImplicitArray”:false,

Usually used when we are moving from a javascript codebase to a typescript codebase where there are generally thousands of functions

5. **removeComments**

Weather or not to include comments in the final js file.

No developer spend time in **js** file hence no sense in bloating it up with comment

“removeComments”: true;

Interfaces

What are interfaces??

How can you assign types to objects? For example, a user object that look like this –

```
const user = {
    firstName: "harkirat",
    lastName: "singh",
    email: "email@gmail.com".
    age: 21,
}
```

To assign a type to the user object, you can use interfaces

```
interface User {
    firstName: string;
    lastName: string;
    email: string;
    age: number;
```

```
}
```

Assignment 1:

Create a function isLegal that return true or false if a user is above 18. It takes a user as an input.

```
function isLegal(user):boolean{
    // what is type of user??
    if(user.age>18){
        return true;
    }else{
        return false;
    }
}
```

What is the type of user??

One way is such

```
function isLegal(user:{
    firstName: string,
    lastName: string,
    age:number
}):boolean{
    // what is type of user??
    if(user.age>18){
        return true;
    }else{
        return false;
    }
}
isLegal({
    firstName:"Nishu",
    lastName:"Thapa",
    age:22
})
```

The problem is that if we have to use this same type in another function example is Greet() then we have to define the same exact type in two places.

Hence we can create an interface how user look like

```
interface User{
    firstName:string,
    lastName:string,
    age:number
}

function isLegal(user:User
):boolean{
    // what is type of user??
    if(user.age>18){
        return true;
    }else{
        return false;
    }
}
function greet(user:User){
    console.log("hello there "+user.firstName)
}

isLegal({
    firstName:"Nishu",
    lastName:"Thapa",
    age:22
})
```

```
email?:string;
```

Denote that email is optional argument user may or may not provide it.

Assignment 2:

Create a React component that takes todos as an input and renders them

```
import './App.css'

function App() {
  return (
    <>
      <Todo
```

```
      title='Go to Gym Bawee'
      description='Bawee chalte hai kal subah'
      done={false} />
    </>
  )
}

interface TodoProp{
  title:string,
  description:string,
  done:boolean
}

function Todo(props: TodoProp){
  return <div>
    <h1>
      {props.title}
    </h1>
    <h3>
      {props.description}
    </h3>

  </div>
}

export default App
```

If we try to give something extra it will raise error.

Another approach

```
// Todo.tsx
interface TodoType {
  title: string;
  description: string;
  done: boolean;
}

interface TodoInput {
```

```
  todo: TodoType;
}

function Todo({ todo }: TodoInput) {
  return <div>
    <h1>{todo.title}</h1>
    <h2>{todo.description}</h2>

  </div>
}
```

Implementing interfaces

Interfaces have another special property. You can implement interfaces as a class.

Let's say you have an person **interface -**

```
interface Person {
  name: string;
  age: number;
  greet(phrase: string): void;
}
```

You can create a class which implements this interface

```
interface Person {
  name: string;
  age: number;
  email?:string;
  greet(phrase: string): void;
}

class Employee implements Person {
  name: string;
  // can be public private
  age: number;
  email?: string;
  constructor(n: string, a: number) {
      this.name = n;
      this.age = a;
  }
```

```
  greet(phrase: string) {
      console.log(`${phrase} ${this.name}`);
  }
}

const e = new Employee("thapa",22);
console.log(e.name)
// Suppose we don't want to mention certian parameter in constructor then
we will make it optional
```

This is useful since now we can have multiple variants of a person (Manager, CEO) .

This is rarely used , it is popular difference between interface and types.

If we define interfaces then we can define a class which follow all three of these properties.

Types doesn't let us do it.'

The whole goal of typescript is to find compile time errors.

Q/Na

1. Zod is for runtime type checks(99%)

Zod does type checking during run time our code is running, our code is executing on browser or node.js

Typescript type checking during compile time.

2. Why even use implement , why don;t we simply define class??

```
class Employee implements Person {
```

It is because suppose Employee implements Person now it has to forceful will also contain the greet function defined in the interface of the person.

```
class Employee implements Person {
```

```
  name: string;
  // can be public private
  age: number;
  email?: string;
  constructor(n: string, a: number) {
      this.name = n;
      this.age = a;
  }
  greet(phrase: string) {
      console.log(`${phrase} ${this.name}`);
  }
}
```

If it doesn't have greet it will give compile time error and ts file wont be converted to the js file.

All these interfaces and types wont be present in the final js file.

Types

What are types??

Very similar to interfaces, types let you **aggregate** data together.

```
type User = {
    firstName: string;
    lastName: string;
    age: number
}
```

We need to put **=** in types but we dont put this in interfaces.

Types cannot be used to implement classes.

Types and Interfaces are mostly same.

Use in general interfaces unless there is need for Types.

But they let us do a few more things.

1. **Unions**

Lets say we want to print the **id of a user, which can be a number or a string**

```
function greet (id:number){

}
greet(1);
greet("1");
```

How to make this work currently it shows error.

Method 1

```
function greet (id:(number | string)){

}
greet(1);
greet("1");
```

Method 2

```
type GreetArg = number | string;

function greet (id:GreetArg){

}
greet(1);
greet("1");
```

Solutipn

```
type StringOrNumber = string | number;

function printId(id: StringOrNumber) {
  console.log(`ID: ${id}`);
}

printId(101); // ID: 101
printId("202"); // ID: 202
```

We can not do this using interfaces.

2. **Intersection**

What if you want to create a type that has every property of multiple types/interfaces

```
type Employee = {
    name: string;
    startDate: Date;
  };

  type Manager = {
    name: string;
    department: string;
  };

  type TeamLead = Employee & Manager;

  const teamLead: TeamLead = {
    name: "harkirat",
    startDate: new Date(),
    department: "Software developer"
  };
```

What is difference between interfaces and types??

Arrays in TS

If you want to access arrays in a typescript , it's as simple as adding a [] annotation next to the type.

Example 1:

Given an array of positive integers as input, return the maximum value of the array

Solution:

type numberArray = number[];

function maxValue(arr: numberArray){

}

```
function maxValue(arr: number[]) {
  let max = 0;
  for (let i = 0; i < arr.length; i++) {
      if (arr[i] > max) {
          max = arr[i]
      }
  }
  return max;
}

console.log(maxValue([1, 2, 3]));
```

Example 2:

Given a list of users, filter out the users that are legal (greater than 18 years of age)

```
interface User {
  firstName: string;
  lastName: string;
  age: number;
}
```

Solution:

```
interface User {
  firstName: string;
  lastName: string;
  age: number;
}

function filteredUsers(users: User[]) {
    return users.filter(x => x.age >= 18);
}
```

```
console.log(filteredUsers([{
    firstName: "harkirat",
    lastName: "Singh",
    age: 21
}, {
    firstName: "Raman",
    lastName: "Singh",
    age: 16
}, ]));
```

Enums

Enums (enumeration) in TypeScript are a feature that allow you to define a set of **named constants**.

The concept behind an enumeration is to create a human-readable way to represent a set of constant values, which might otherwise be represented as numbers or strings.

Example 1 - Game

Let's say you have a game where you have to perform an action based on weather the user has pressed the up arrow key, down arrow key, left arrow key or right arrow key.

```
function doSomething( keyPresed){
        // do something
}
```

What should the type of keyPressed be?

Should it be a string? (UP , DOWN , LEFT, RIGHT) ?

Should it be numbers? (1, 2, 3, 4) ?

```
type KeyInput = "up" | "down" |"left" |"right"
// now it will show error
// this already looks human readable

function doSomething( keyPresed: KeyInput){
    // do something
    if( keyPresed == "up"){

    }
}

doSomething("up")
doSomething("down")
// doSomething("downRandomsG")
// now this shows errors
// due to this random value our logic will fail
// this would be good if typescript somehow knows how to do it
```

Lets use enums

```
type KeyInput = "up" | "down" |"left" |"right"
// now it will show error
// this already looks human readable

// enumeration
enum Direction {
    Up,    // 0
    Down,  // 1
    Left,  // 2
    Right  // 3
}

function doSomething( keyPresed: Direction){
    // do something
    if( keyPresed == Direction.Up){

    }
}
```

```
doSomething(Direction.Up)
doSomething(Direction.Down)
// doSomething("downRandomsG")
// now this shows errors
// due to this random value our logic will fail
// this would be good if typescript somehow knows how to do it
```

Direction.up is basically 0

Just a virtual concept in typescript. It is not something which Javascript understands

What if i don't want Direction.up want to be 0, 1 ,2 etc

We can write it like this

```
enum Direction {
    Up ="up",
    Down="down",
    Left="left",
    Right="right"
}
```

If we give one of them type them we have to give to all.

We can also start from a number

```
// enumeration
enum Direction {
    Up =1,
    Down,
    Left,
    Right
}
```

We don't ever use constant literal that what enums are for

Common usecase in express

```
enum ResponseStatus {
    Success = 200,
    NotFound = 404,
```

```
    Error = 500
}

app.get("/', (req, res) => {
    if (!req.query.userId) {
            res.status(ResponseStatus.Error).json({})
    }
    // and so on...
        res.status(ResponseStatus.Success).json({});
})
```

```
const app = express()

enum ResponseStatus {
    Success = 200,
    NotFound = 404,
    Error = 500
}

app.get("/', (req, res) => {
    if (!req.query.userId) {
            res.status(ResponseStatus.Error).json({})
    }
    // and so on...
        res.status(ResponseStatus.Success).json({});
})
// also benefit if we have multiple http server
// and all we need to do change in a single place
```

Generics

Generics are a language independent concept

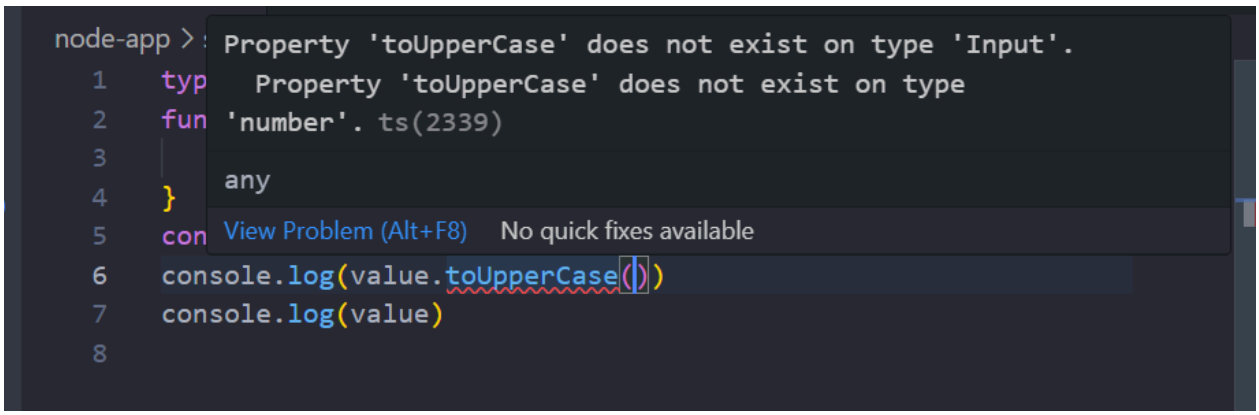
Examples

1. Problem Statement

Let's say you have a function that needs to return the first element of an array.
Array can be of type either string or integer.
How would you solve this problem?

```
type Input = number | string;
function firstEl(arr: Input[]){
    return arr[0];
}
const value = firstEl(["thapa","Nishu"]);
console.log(value)
```

What is problem in this approach ? Lets say we want to print final thing in uppercase.
If typescript was smart it would have guessed that value in string.
But it raises error when we try to uppercase
Says its not necessary it will be string. The problem is typescript can't run any code.



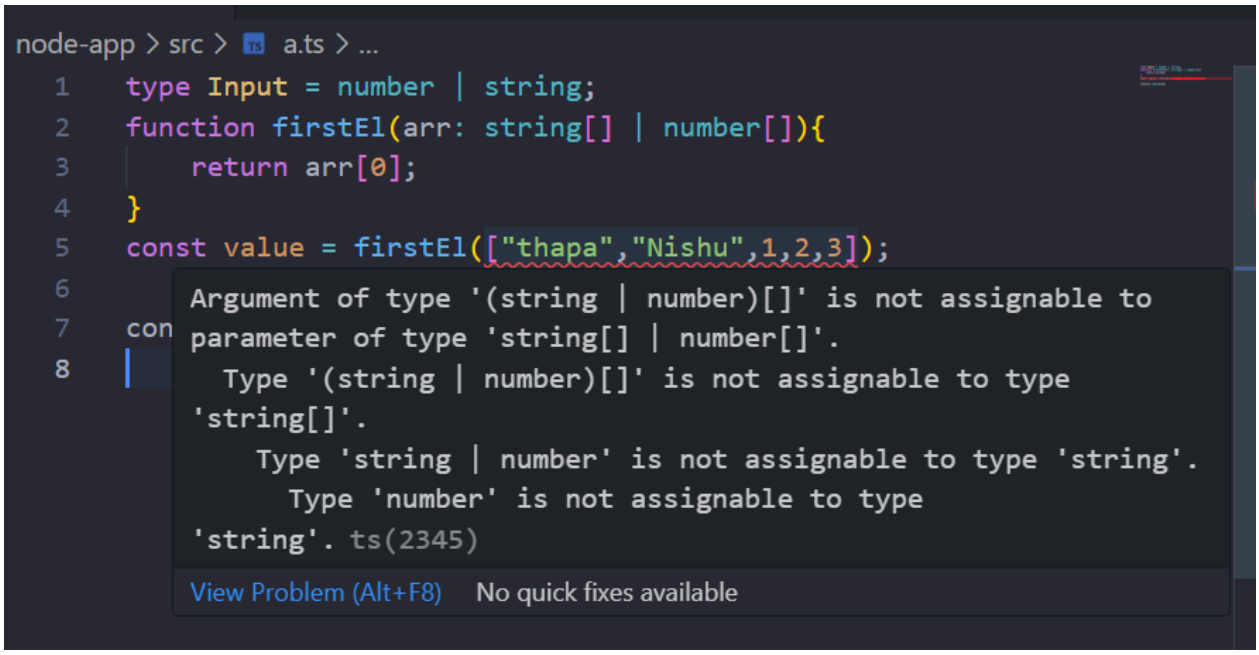
Another problem is that user can send both number and string

```
type Input = number | string;
function firstEl(arr: Input[]){
    return arr[0];
}
const value = firstEl(["thapa","Nishu",1,2,3]);
```

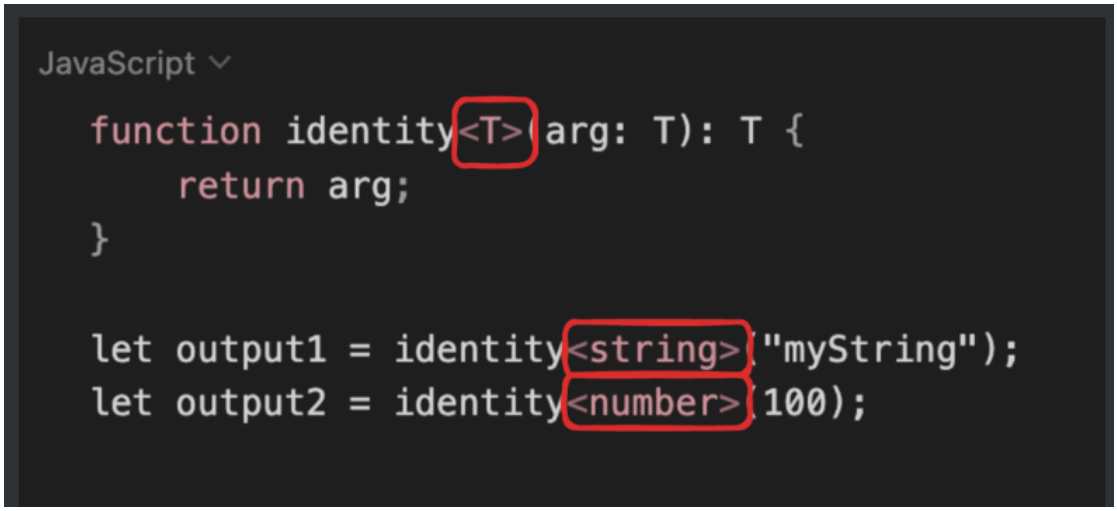
```
console.log(value)
```

We can see that typescript doesn’t complain

We can fix it lets see



Generics enable you to create components that work with any data type while still providing compile- time type safety



```
// input argument is generic and can be anything, whoever is calling the
function will tell
function identity<T>(arg: T){
    return arg
}
let output1 = identity<string>("myString");
let output2 = identity<number>(100);
// we can also think that we have created two variant of our function
```

```
function getFirstElement<T>(arr: T[]):T {
    return arr[0];
}

interface User{
    name: string;
}

const el = getFirstElement<User>([{name: "thapa"}]);
// const el = getFirstElement(["harkiratSingh", "ramanSingh"]);
// console.log(el.toLowerCase())
const el2 = getFirstElement([1,2])
```

Exporting an importing modules

TypeScript follows the ES6 module system, using import and export statements to share code.

Until now we have been following this syntax

const express = require(“express”)

1. **Constant exports**

npm install express @types/express

```
import express from "express"
```

export const a = 1;

math.ts

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

export function subtract(x: number, y: number): number {
    return x - y;
}
```

main.ts

```
import { add } from "./math"

add(1, 2)
```

2. Default Exports

```
export default class Calculator {
    add(x: number, y: number): number {
        return x + y;
    }
}
```

calculator.ts

```
import Calculator from './Calculator';

const calc = new Calculator();
console.log(calc.add(10, 5));
```