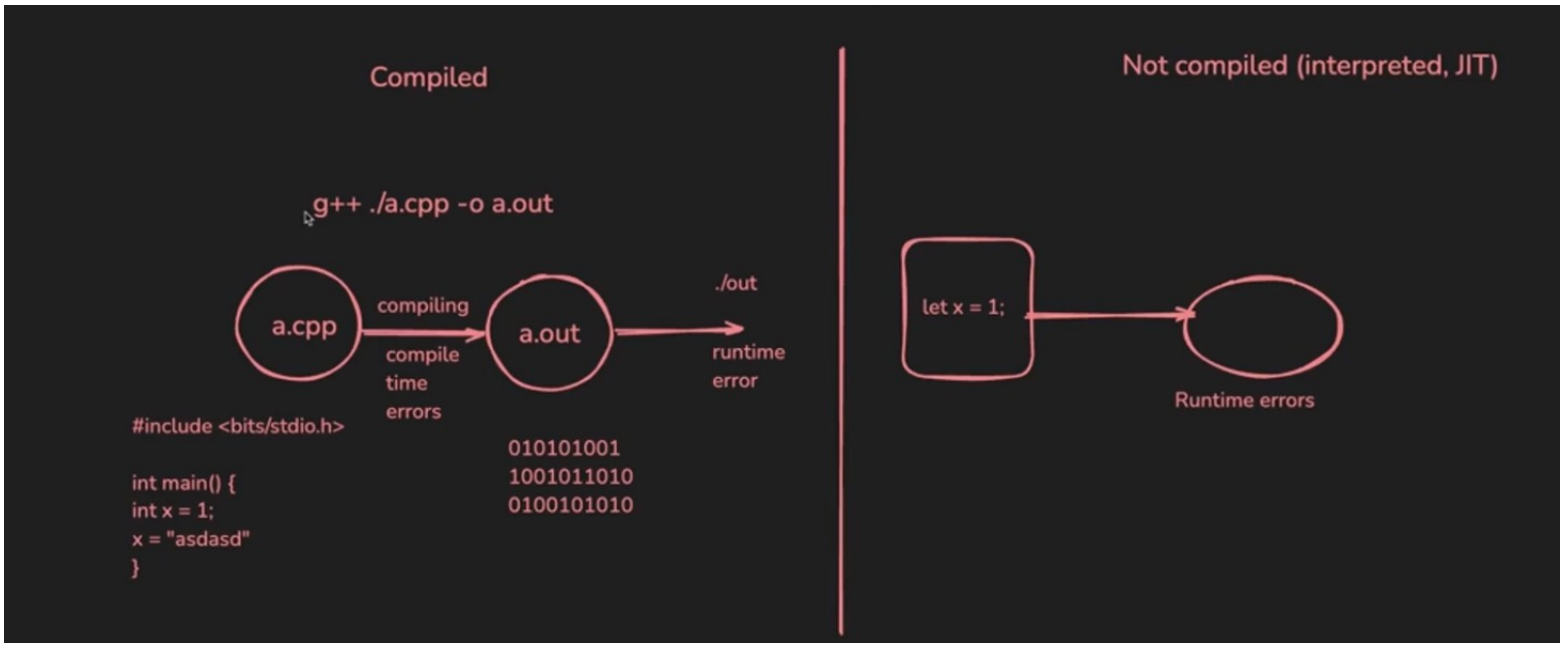
Typescript



1. Strongly typed vs loosely typed

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

Strongly typed languages

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

Loosely typed languages

1. Examples - Python, Javascript, Perl, php
2. Benefits
    1. Easy to write code
    2. Fast to bootstrap
    3. Low learning curve

Code doesn't work ❌

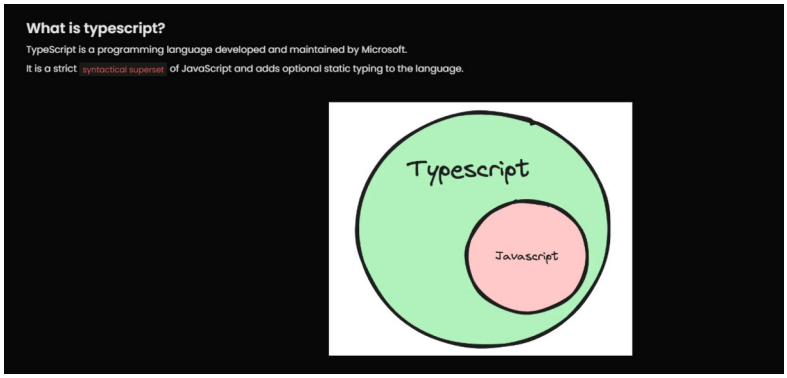
```
#include <iostream>

int main() {
 int number = 10;
 number = "text";
 return 0;
}
```

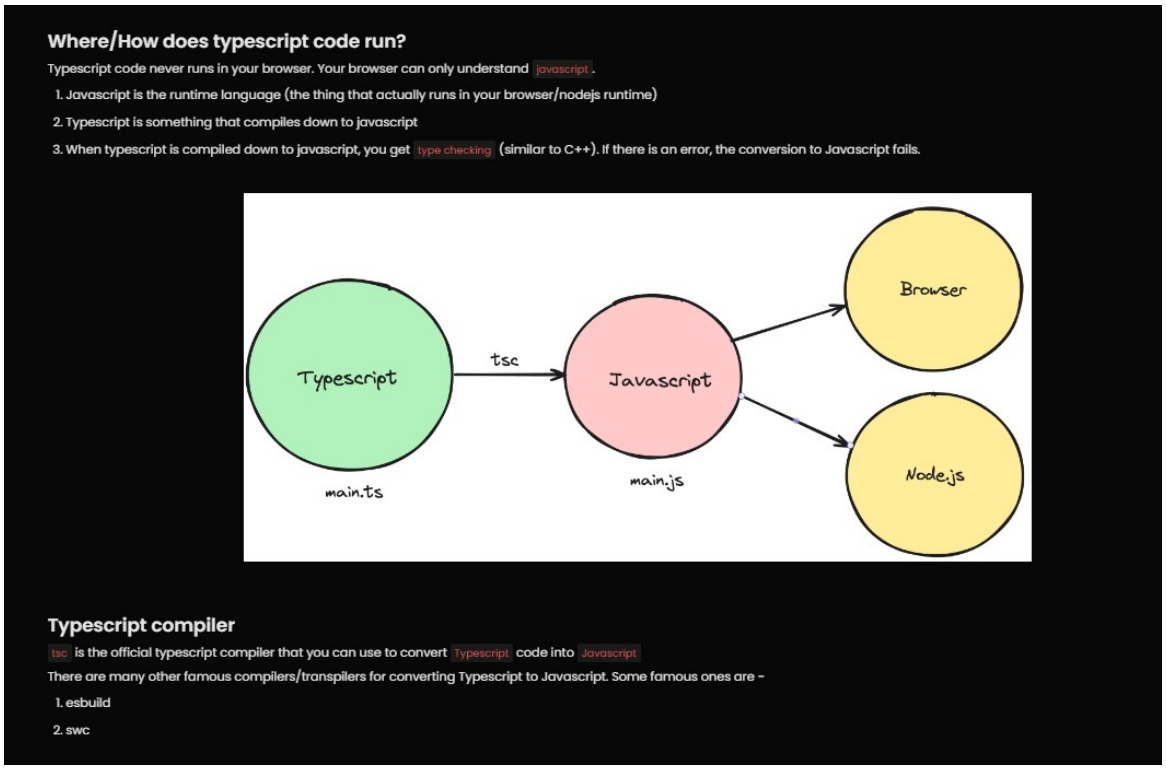
Code does work ✅

```
function main() {
 let number = 10;
 number = "text";
 return number;
}
```

People realised that javascript is a very power language, but lacks types. `Typescript` was introduced as a new language to add `types` on top of javascript.

Javascript is a loosely typed language to make it strongly so that the errors in the code can be solved in compile only for that the devs made **TYPESCRIPT** it just js but with types





Step to create a ts project:

npm install -g typescript (to globally install the ts)

Step 1: type npm init-y

Step 2: npm install typescript

npm install -d typescript

Step 3: npx tsc -–init (a exec file to initialize typescript)\

In the file un comment the rootDir and outDir line

Step 4: create a index.ts file

To execute the ts file

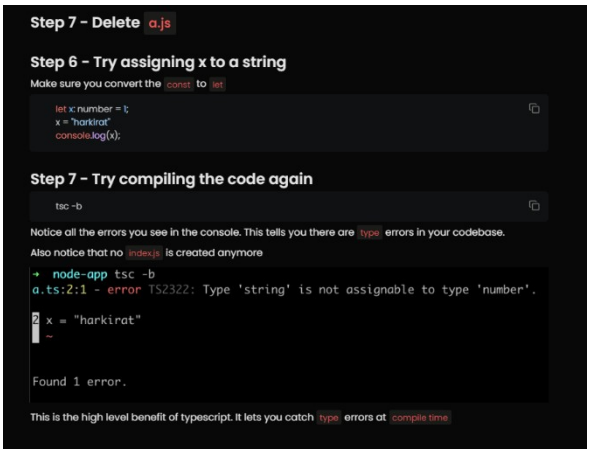
Step 1: run npx tsc-b // a index.js file will be created and just run that

Step 2: run node index.js

In ts if we have written

Let x = 1; // ts did type inferencing to identify whether it’s a number or not its like assumed it’s a number and every time it does not do that

TS FILES NEVER RUNS SOLLELY IT FIRST CONVERTS TO JS AND THEN RUNS



let number:string = “Modi”;

in ts there is a special type

let a:any = 1;

a = “wow”;

you can define anything then

```
function greet(name:string | number){
    console.log("Hello " + name);
}

greet("Modi");
greet(1);
```

To explicitly tell whether it’s a number or not

```
function sum(a:number,b:number):number{
    return (a+b);
}
let ans = sum(10,2);
console.log(ans);
```

ans is explicitly defined number then

it can automatically refer also that it’s a number which is called **type inferencing**

```
function delayedCall(anotherFn: () => void){
    setTimeout(anotherFn,1000);
}

function log(){
    console.log("hi there");
}
delayedCall(log)
```

In this the below function is defined and the

(Fn: () => void) means

Niche walae function mai koi argument nhi hai and void return karta hai

ECMASCRIPT gives the version of the js and brings new syntax and all like arrow fn, const, let etc

For this only internet explorer stopped working because they never adapted the newer version of js

Chrome js compiler = v8 engine

Firefox js compiler = spidermonkey engine

Tsconfig file

1. Target = it helps to fix the ecmascript version

2. rootDir = one is src folder and dist folder src folder contain ts files and the src folder contains js files. Helps to structure the files and only the required files will be pushed to github later than.
3. ImplicitAny = this is to remove the red underline of the variable we pass in the function and not defined its type
   eg name: number no error
   name red underline

Primitive and non primitive types

String , Boolean, number etc are primitive

Interfaces, types are all non primitive

Interfaces

```ts
interface User {
    name:string;
    age:number;
    address:{
        city:string;
        country:string;
        pincode:number;
    },
};
let user: User ={
    name:"harkirat",
    age:21,
    address:{
        city:"Chandigarh",
        country:"India",
        pincode: 15605,
    },
};
```

```ts
function isLegal(user: User): boolean{
    if(user.age >=18){
        return true;
    }
    else{
        return false;
    }
}
const ans = isLegal(user);
if(ans) {
    console.log("I am legal");
}else{
    console.log("I am illegal")
}
```

Problem with interfaces is that no column should be left unfilled bcoz it will cause error if the column is not filled.

To make a specific key optional you have to add (address?:) to make it optional

One interface can use another interface eg:

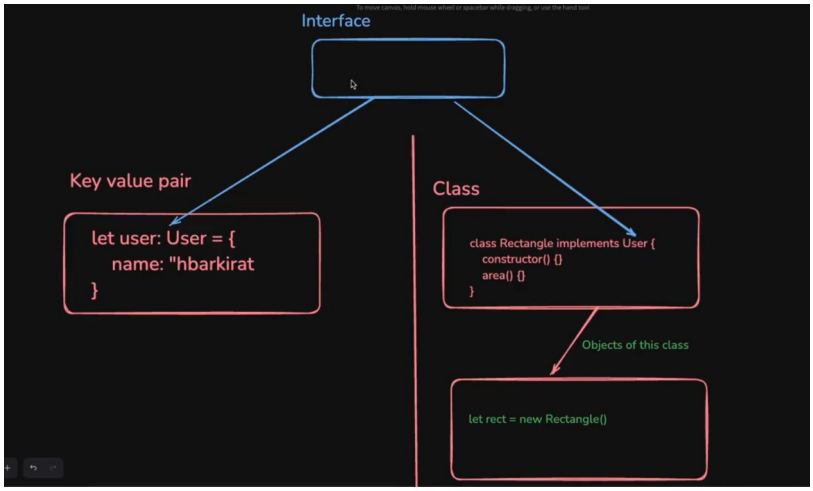
```ts
interface Address{
    city:string;
    country:string;
    pincode:number;
}
interface User {
    name:string;
    age:number;
    address:Address
};
```

The interface address is used by another interface user for the address thing. This will reduce the redundancy of using the same code.

Passing a function to interface

```ts
interface People{
    name:string,
    age:number,
    greet: () => string,
}
let user:People ={
    name:"harkirat",
    age: 21,
    greet: () => {
        return "hi";
    }
}
let ans = console.log(user.greet());
console.log(ans)
```

Implements in interfaces



Constructor is used when we make a object firstly the constructor is called then other functions is done

Extra things can be added in the constructor but the mentioned things in interface must be there

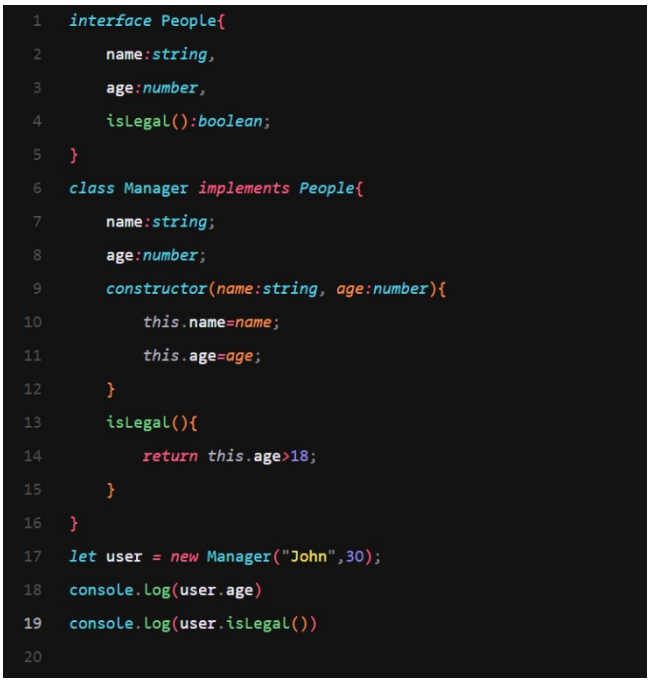
```ts
interface People{
    name:string,
    age:number,
    // greet: () => string,
}
class Manager implements People{
```

```ts
class Shape {
  constructor() {}
  area() {
    console.log("hi i am area");
  }
}

class Rectangle extends Shape {
  width: number;
  height: number;

  constructor() {
    super()
    this.width = 1;
    this.height = 2;
  }
}
```

Extends and implements are different because extends means you are passing eg area function from shape to cuboid class directly no need to define anything.

Super() is also required when you are calling something from parent class to just initialize it

Implements help in making function inside which can later be called also.



```
interface People{
    name:string,
    age:number,
    isLegal():boolean;
}
class Manager implements People{
    name:string;
    age:number;
    constructor(name:string, age:number){
        this.name=name;
        this.age=age;
    }
    isLegal(){
        return this.age>18;
    }
}
let user = new Manager("John",30);
console.Log(user.age)
console.Log(user.isLegal())
```

Use of implements and Class:

Maybe in future need to create a new class then you have to write the whole name, age all inside that but in this you can just extends the code and use the manager class directly.

DIFFERENCE IN INTERFACE AND TYPES IS IN INTERFACE YOU CAN IMPLEMENT CLASS BUT YOU CANNOT DO THAT IN TYPES

DIFFERENCE IN ABSTRACT AND INTERFACE IS YOU CAN CREATE A DEFAULT IMPLEMENTATION IN ABSTRACT BUT NOT IN INTERFACE.

```
abstract class User {
  name: string;
  constructor(name: string) {
    this.name = name;
  }

  abstract greet(): string;
  hello() {
    console.log("hi there");
  }
}

class Employee extends User {
  name: string;
  constructor(name: string) {
    super(name)
    this.name = name;
  }
  greet() {
    return "hi " + this.name;
  }
}
```

```
interface User2 {
  name: string
}

type User = {
  name: string;
  age: number
}

function isLegal(user: User) {
  return user.age > 18
}
```

Types

Its same as interface just the difference in how you define it like in types you have to add a equal sign

1. Types are better as we can do intersection and union of the different eg teamlead can have properties of both employee and manager.
2. Types cannot be implemented by the class

Arrays

```ts
function getMax(nums: number[]) { 8 hidden lines }

getMax([1, 2, 3])

interface Address {
  city: string;
  pincode: string;
}


interface User {
  name: string;
  age: number;
  addresses: Address[];
}

let user: User = {
  name: "harkirat",
  age: 21,
  addresses: []
}
```

```ts
interface User{
    firstName:string;
    LastName:string;
    age:number;
}
function ageLegal(user:User[]){
    let ans = []
    for(let i=0;i<user.Length;i++){
        if (user[i]!.age > 18){
            ans.push(user[i]);
        }
    }
    return ans;
}
const user:User[] = [
    {firstName:"Alice",LastName:"smthg",age:30},
    {firstName:"Malice",LastName:"smthg",age:3},
    {firstName:"talice",LastName:"smthg",age:21},
]
console.log(ageLegal(user))
```

DIFFERENCE BETWEEN UNION AND INTERSECTION

Remember that ts is not sealed and is a open type

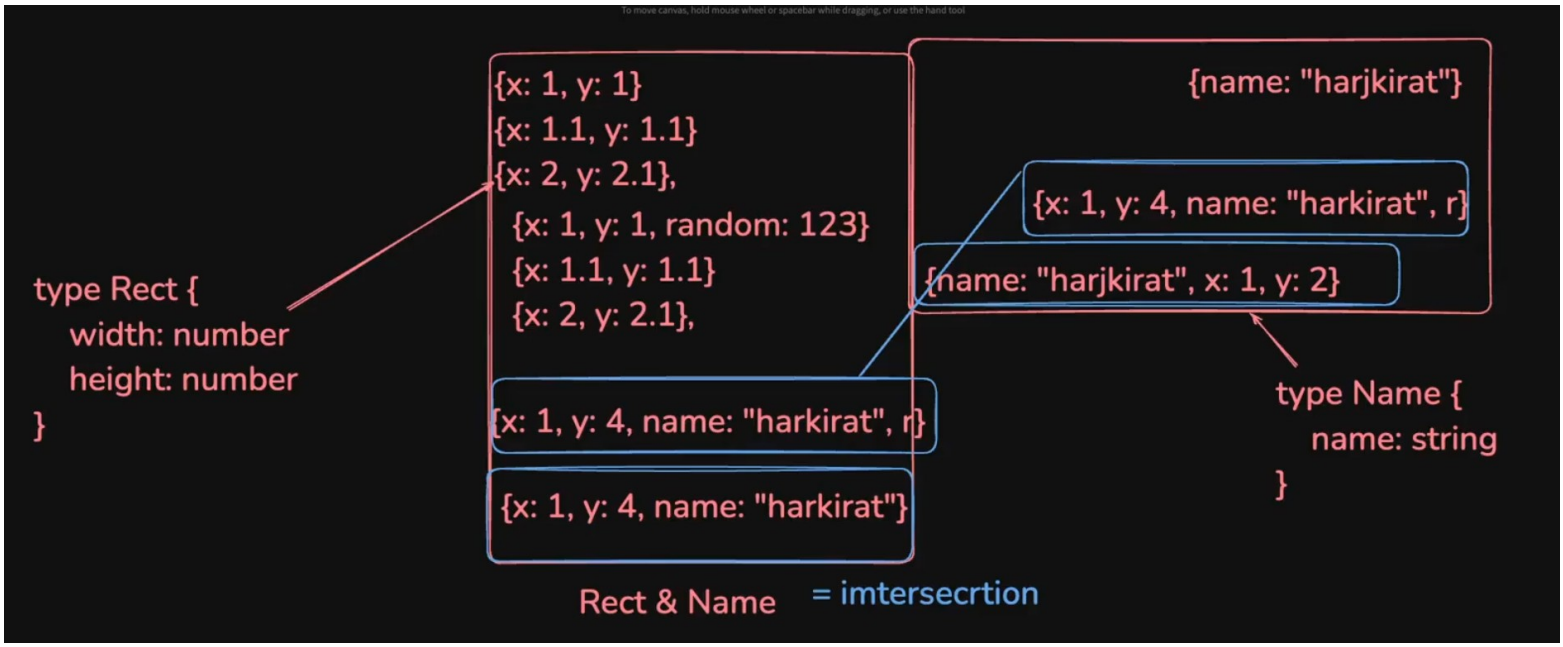
INTERSECTION

Isme joh elements ka set hai usme inf sets hai jaise rect mai width and height hai but usme w,h toh hona hii chaiye but w,h , name bhi chalega that’s the catch in this.

Toh rect mai width, height and name ho skta hai and

Name mai name, width and height ho skta hai

Toh isme common toh dono ki sari property hui which seems like UNION but its intersection



UNION

Isme mai toh sabh hai but agar width hai toh height bhi hona chaiye aek set kae sarae elements hone chiaye

Link to notes

Advance ts APIs

1. **Pick**: Let you pick set of properties from the interface such as name, email, password etc.

```ts
interface User{
    id:string;
    name:string;
    age:number;
    email:string;
    password:string;
}
type UpdateProps = Pick<User, 'name' | 'age' | 'email'>
function updateUser(updatedProps: UpdateProps){
    console.log(`Name: ${updatedProps.name}, Email: ${updatedProps.email}`)
}

```

2. **Partial:** Make the values you give optional

```ts
interface User{
    id:string;
    name:string;
    age:number;
    email:string;
    password:string;
}
type UpdateProps = Pick<User, 'name' | 'age' | 'email'>
type UpdatePropsOptional = Partial<UpdateProps>
function updateUser(updatedProps:UpdatePropsOptional){

}
updateUser({
    email:"123"
})
```

3. In js and ts both you can change the values inside a array or a object to fix that you cannot reassign the whole array but you can reassign the inside thing.
   To force that neither the internal value change nor the whole object you use this
   Readonly: Lets you just read the value you cannot change it

```ts
interface User{
    id:string;
    name:string;
    age:number;
    email:string;
    password:string;
}
type UpdateProps = Pick<User, 'name' | 'age' | 'email'>
type UpdatePropsOptional = Partial<UpdateProps>
function updateUser(updatedProps:UpdatePropsOptional){

}
updateUser({
    email:"123"
})
```

Used in like making the apikey read only so that the developer by mistakenly does not change the value of the api key

```ts
type User ={
    id:string;
    username:string;
}
type Users = {
    [key:string]:User;
}
const users:Users = {
    "ras@qd1" : {
        id: 'ras@qd1',
        username: 'harkirat'
    },
    "ras@qd2" : {
        id: 'ras@qd2',
        username: 'raman'
    },
}

```

Ugly way to represent the code

To fix that ts introduced 2 new things

4. Record: it's a ts thing Cleaner way to create objects

```ts
type Users = Record<string, number>;
const users: Users = {
    "abc": 21,
    "bcd": 22
}
```

```ts
type Users = Record<string, {age: number;name:string}>;
const users: Users = {
    "abc": {age:21, name:"harkirat"},
    "bcd": {age:33,name:"abc"}
}
users["abc"];
```

5. Map: It's a js concept – USE THIS MORE

```ts
const users = new Map()
users.set("abc", {name:"abc1", age:30,email:"somthg@gmail.com"})
users.set("bcd", {name:"bcd1", age:31,email:"somthg1@gmail.com"})
const user = users.get("abc")

```

```ts
type User = {
    name:string;
    age: number;
    email:string;
}
const users = new Map<string, User>()
users.set("abc", {name:"abc1", age:30})
users.set("bcd", {name:"bcd1", age:31,email:"somthg1@gmail.com"})
const user = users.get("abc")
```

In this now it complains also if there is any problem as we have defined what there should be in the map.

6. Exclude

```ts
type EventType = 'click' | 'scroll' | 'mousemove';
type ExcludeEvent = Exclude<EventType, 'scroll'>; // picks 'click' | 'mousemove'

const handleEvent = (event: ExcludeEvent) => {
  console.log(`Handling event: ${event}`);
};

handleEvent('click'); // OK
handleEvent('scroll');
```

Removes the word which we never wants

ZOD

```ts
import { z } from 'zod';
import express from "express";

const app = express();

// Define the schema for profile update
const userProfileSchema = z.object({
  name: z.string().min(1, { message: "Name cannot be empty" }),
  email: z.string().email({ message: "Invalid email format" }),
  age: z.number().min(18, { message: "You must be at least 18 years old" }).
  optional(),
});
type FinalUserSchema = z.infer<typeof userProfileSchema>;
// the above thing helps to get the structure of the userProfileSchema without
re defining it
app.put("/user", (req, res) => {
  const { success } = userProfileSchema.safeParse(req.body);
  const updateBody:FinalUserSchema = req.body;

  if (!success) {
    res.status(411).json({});
    return
  }
  // update database here
  res.json({
    message: "User updated"
  })
});
app.listen(3000);
```

Infer thing is imp