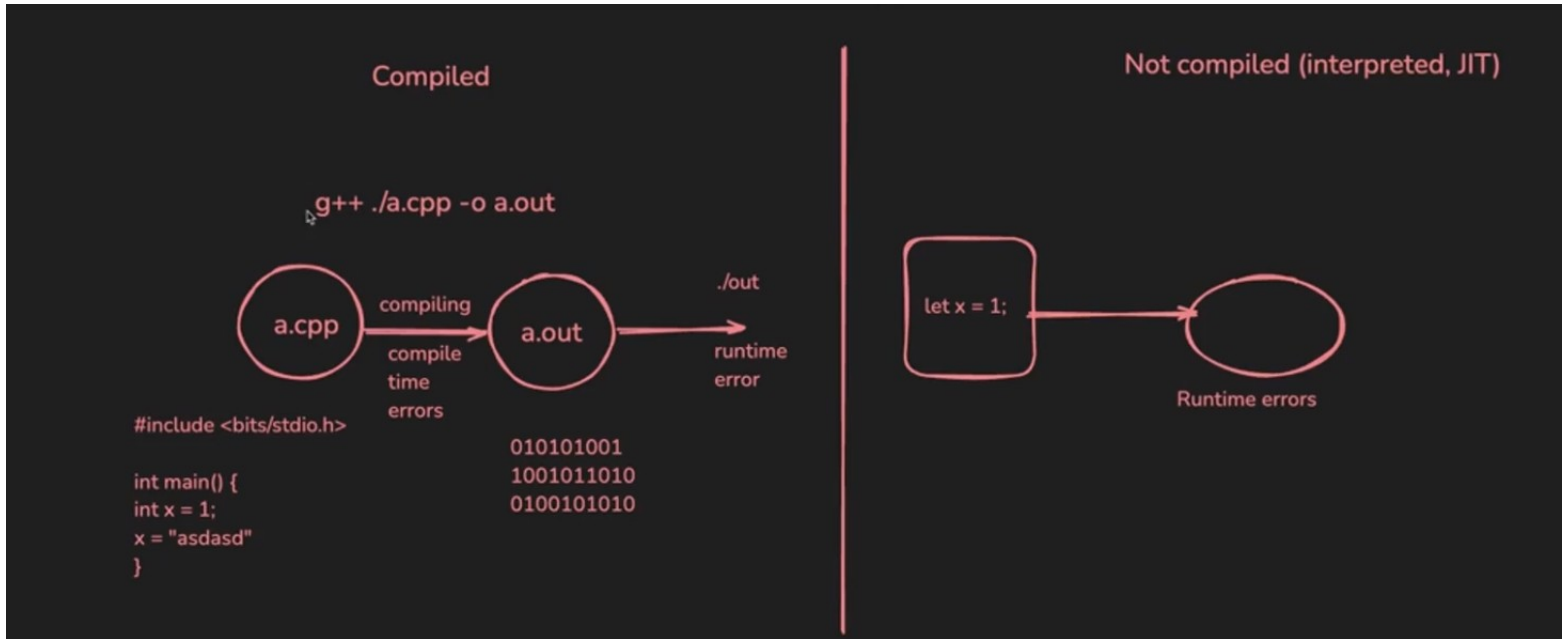


# Typescript



## 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>

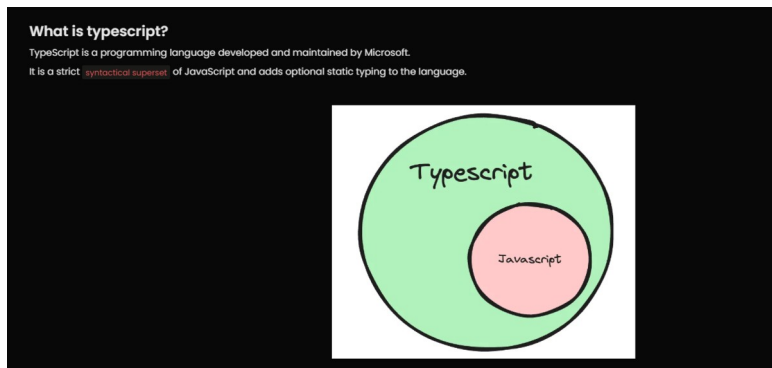
int main() {
  int number = 10;
  number = "text";
  return 0;
}
```

### 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.

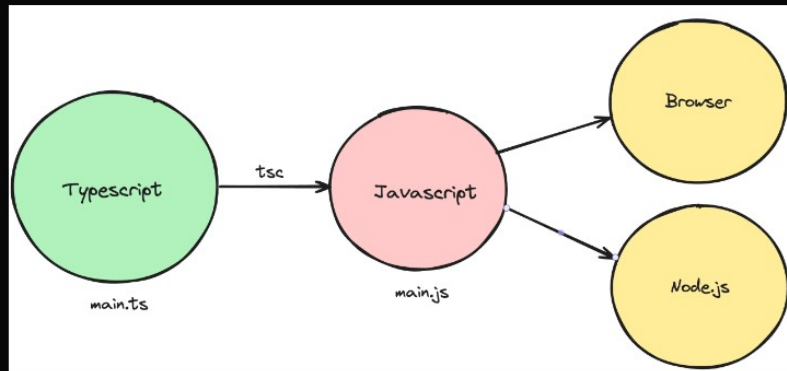
Javascript is a loosely typed language to make it strongly so that the errors in the code can be solved in compile only for that the devs made **TYPESCRIPT** it just js but with types



### Where/How does typescript code run?

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

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



### Typescript compiler

`tsc` is the official typescript compiler that you can use to convert `Typescript` code into `Javascript`.

There are many other famous compilers/transpilers for converting Typescript to Javascript. Some famous ones are -

1. esbuild
2. swc

### 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

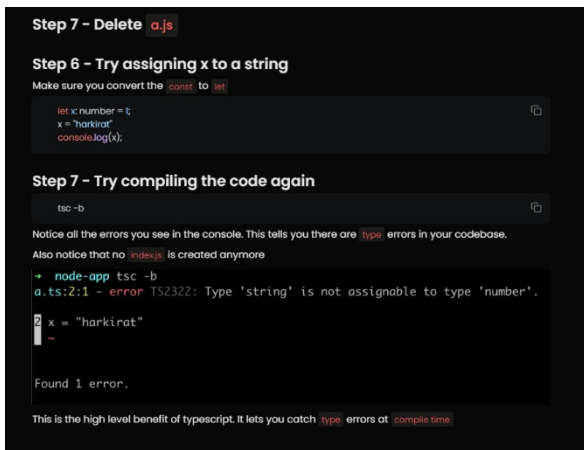
Step 1: run `npx tsc-b //` a `index.js` file will be created and just run that

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

```

1 function greet(name:string | number){
2     console.log("Hello " + name);
3 }
4
5 greet("Modi");
6 greet(1);
  
```

To explicitly tell whether it's a number or not

```

1 function sum(a:number,b:number):number{
2     return (a+b);
3 }
4 let ans = sum(10,2);
5 console.log(ans);
1 function delayedCall(anotherFn: () => void){
2     setTimeout(anotherFn,1000);
3 }
4
5 function log(){
6     console.log("hi there");
7 }
8 delayedCall(log)
  
```

ans is explicitly defined number then  
 it can automatically refer also that it's a  
 number which is called **type inferencing**

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

```

1 interface User {
2     name:string;
3     age:number;
4     address:{
5         city:string;
6         country:string;
7         pincode:number;
8     },
9 };
10 let user: User =({
11     name:"harkirat",
12     age:21,
13     address:{
14         city:"Chandigarh",
15         country:"India",
16         pincode: 15605,
17     },
18 };|

```

```

19 function isLegal(user: User): boolean{
20     if(user.age >=18){
21         return true;
22     }
23     else{
24         return false;
25     }
26 }
27 const ans = isLegal(user);
28 if(ans) {
29     console.log("I am legal");
30 }else{
31     console.log("I am illegal")
32 }
33

```

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:

```

1 interface Address{
2     city:string;
3     country:string;
4     pincode:number;
5 }
6 interface User {
7     name:string;
8     age:number;
9     address:Address
10 };

```

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

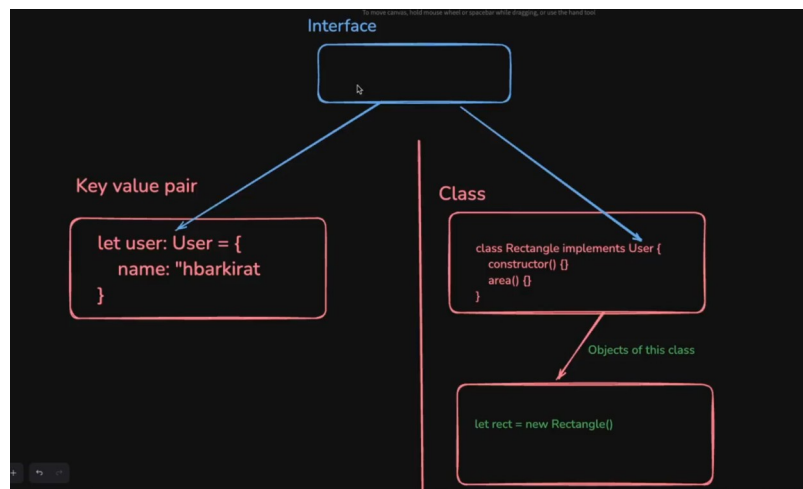
Passing a function to interface

```

1 interface People{
2     name:string,
3     age:number,
4     greet: () => string,
5 }
6 let user:People ={
7     name:"harkirat",
8     age: 21,
9     greet: () => {
10         return "hi";
11     }
12 }
13 let ans = console.log(user.greet());
14 console.log(ans)
15 |

```

## 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

```

1 interface People{
2     name:string,
3     age:number,
4     // greet: () => string,
5 }
6 class Manager implements People{
7
8 }
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

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.

```
1 interface People{
2     name:string,
3     age:number,
4     isLegal():boolean;
5 }
6 class Manager implements People{
7     name:string;
8     age:number;
9     constructor(name:string, age:number){
10        this.name=name;
11        this.age=age;
12    }
13    isLegal(){
14        return this.age>18;
15    }
16 }
17 let user = new Manager("John",30);
18 console.log(user.age)
19 console.log(user.isLegal())
20
```

### 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.**

```
2 abstract class User {
3     name: string;
4     constructor(name: string) {
5         this.name = name;
6     }
7
8     abstract greet(): string;
9     hello() {
10        console.log("hi there");
11    }
12 }
13
14 class Employee extends User {
15     name: string;
16     constructor(name: string) {
17         super(name)
18         this.name = name;
19     }
20     greet() {
21         return "hi " + this.name;
22     }
23 }
24
```

```
2 interface User2 {
3     name: string
4 }
5
6 type User = {
7     name: string;
8     age: number
9 }
10
11 function isLegal(user: User) {
12     return user.age > 18
13 }
```

## 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

```
1 > function getMax(nums: number[]) { 8 hidden lines }
11
12 getMax([1, 2, 3])
13
14 interface Address {
15   city: string;
16   pincode: string;
17 }
18
19
20 interface User {
21   name: string;
22   age: number;
23   addresses: Address[];
24 }
25
26 let user: User = {
27   name: "harkirat",
28   age: 21,
29   addresses: []
30 }
```

```
1 interface User{
2   firstName:string;
3   LastName:string;
4   age:number;
5 }
6 function ageLegal(user:User[]){
7   let ans = []
8   for(let i=0;i<user.length;i++){
9     if (user[i].age > 18){
10      ans.push(user[i]);
11    }
12  }
13  return ans;
14 }
15 const user:User[] = [
16   {firstName: "ALice", LastName: "smthg", age:30},
17   {firstName: "MaLice", LastName: "smthg", age:3},
18   {firstName: "taLice", LastName: "smthg", age:21},
19 ]
20 console.log(ageLegal(user))
```

## DIFFERENCE BETWEEN UNION AND INTERSECTION

**Remember that ts is not sealed and is a open type**

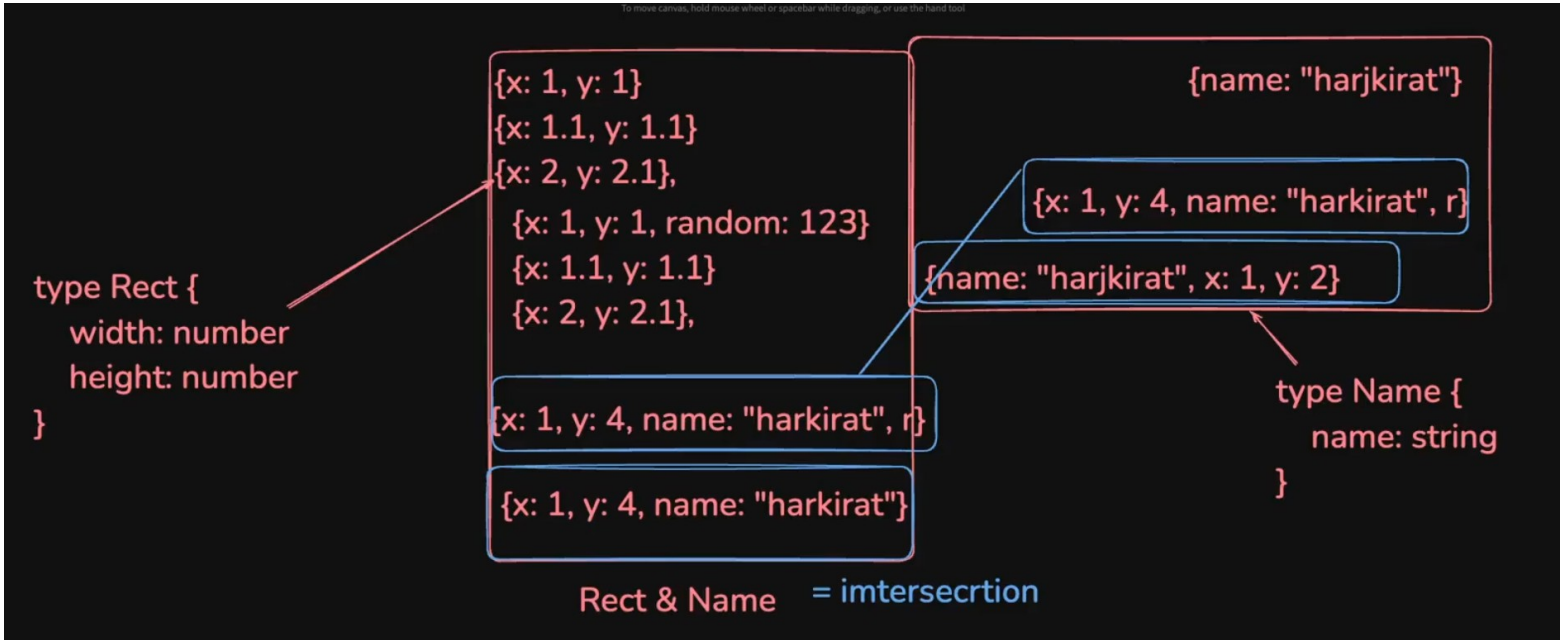
### INTERSECTION

Isme joh elements ka set hai usme inf sets hai jaise rect mai width and height hai but usme w,h toh hona hii chahiye 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 chahiye aek set kae sarae elements hone chahiye

[Link to notes](#)

## Advance ts APIs

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

```

1 interface User{
2   id:string;
3   name:string;
4   age:number;
5   email:string;
6   password:string;
7 }
8 type UpdateProps = Pick<User, 'name' | 'age' | 'email'>
9 function updateUser(updatedProps: UpdateProps){
10   console.Log(`Name: ${updatedProps.name}, Email: ${updatedProps.email}`)
11 }
12

```

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

```

1 interface User{
2   id:string;
3   name:string;
4   age:number;
5   email:string;
6   password:string;
7 }
8 type UpdateProps = Pick<User, 'name' | 'age' | 'email'>
9 type UpdatePropsOptional = Partial<UpdateProps>
10 function updateUser(updatedProps:UpdatePropsOptional){
11
12 }
13 updateUser({
14   email:"123"
15 })

```

## 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

```

1 interface User{
2   id:string;
3   name:string;
4   age:number;
5   email:string;
6   password:string;
7 }
8 type UpdateProps = Pick<User, 'name' | 'age' | 'email'>
9 type UpdatePropsOptional = Partial<UpdateProps>
10 function updateUser(updatedProps:UpdatePropsOptional){
11
12 }
13 updateUser({
14   email:"123"
15 })

```

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

```

1 type User = {
2   id:string;
3   username:string;
4 }
5 type Users = {
6   [key:string]:User;
7 }
8 const users:Users = {
9   "ras@qd1" : {
10     id: 'ras@qd1',
11     username: 'harkirat'
12   },
13   "ras@qd2" : {
14     id: 'ras@qd2',
15     username: 'raman'
16   },
17 }
18

```

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

```
1 type Users = Record<string, number>;
2 const users: Users = {
3   "abc": 21,
4   "bcd": 22
5 }
```

```
1 type Users = Record<string, {age: number; name: string}>;
2 const users: Users = {
3   "abc": {age: 21, name: "harkirat"},
4   "bcd": {age: 33, name: "abc"}
5 }
6 users["abc"];
```

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

```
1 const users = new Map()
2 users.set("abc", {name: "abc1", age: 30, email: "somthg@gmail.com"})
3 users.set("bcd", {name: "bcd1", age: 31, email: "somthg1@gmail.com"})
4 const user = users.get("abc")
5
```

```
1 type User = {
2   name: string;
3   age: number;
4   email: string;
5 }
6 const users = new Map<string, User>()
7 users.set("abc", {name: "abc1", age: 30})
8 users.set("bcd", {name: "bcd1", age: 31, email: "somthg1@gmail.com"})
9 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

```
1 type EventType = 'click' | 'scroll' | 'mousemove';
2 type ExcludeEvent = Exclude<EventType, 'scroll'>; // picks 'click' | 'mousemove'
3
4 const handleEvent = (event: ExcludeEvent) => {
5   console.log(`Handling event: ${event}`);
6 };
7
8 handleEvent('click'); // OK
9 handleEvent('scroll');
```

Removes the word which we never wants

# ZOD

```
1  import { z } from 'zod';
2  import express from "express";
3
4  const app = express();
5
6  // Define the schema for profile update
7  const userProfileSchema = z.object({
8    name: z.string().min(1, { message: "Name cannot be empty" }),
9    email: z.string().email({ message: "Invalid email format" }),
10   age: z.number().min(18, { message: "You must be at least 18 years old" }).
11     optional(),
12 });
13 type FinalUserSchema = z.infer<typeof userProfileSchema>;
14 // the above thing helps to get the structure of the userProfileSchema without
15 // re defining it
16 app.put("/user", (req, res) => {
17   const { success } = userProfileSchema.safeParse(req.body);
18   const updateBody:FinalUserSchema = req.body;
19
20   if (!success) {
21     res.status(411).json({});
22     return
23   }
24   // update database here
25   res.json({
26     message: "User updated"
27   })
28 });
29 app.listen(3000);
```

**Infer thing is imp**