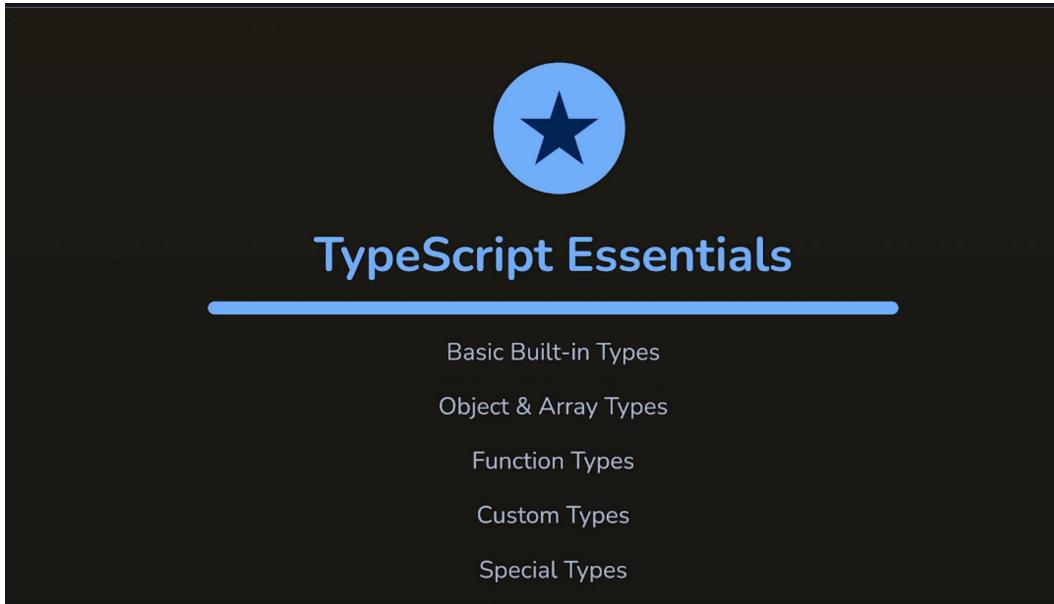


Section 2: TypeScript Basics & Basic Types

22 May 2025 13:10



The slide features a large blue circle with a white star in the center. Below it, the title "TypeScript Essentials" is displayed in a bold, sans-serif font. A horizontal blue bar spans across the middle of the slide. Below the bar is a vertical list of navigation items:

- Basic Built-in Types
- Object & Array Types
- Function Types
- Custom Types
- Special Types

Using Node.js To Run JavaScript Code

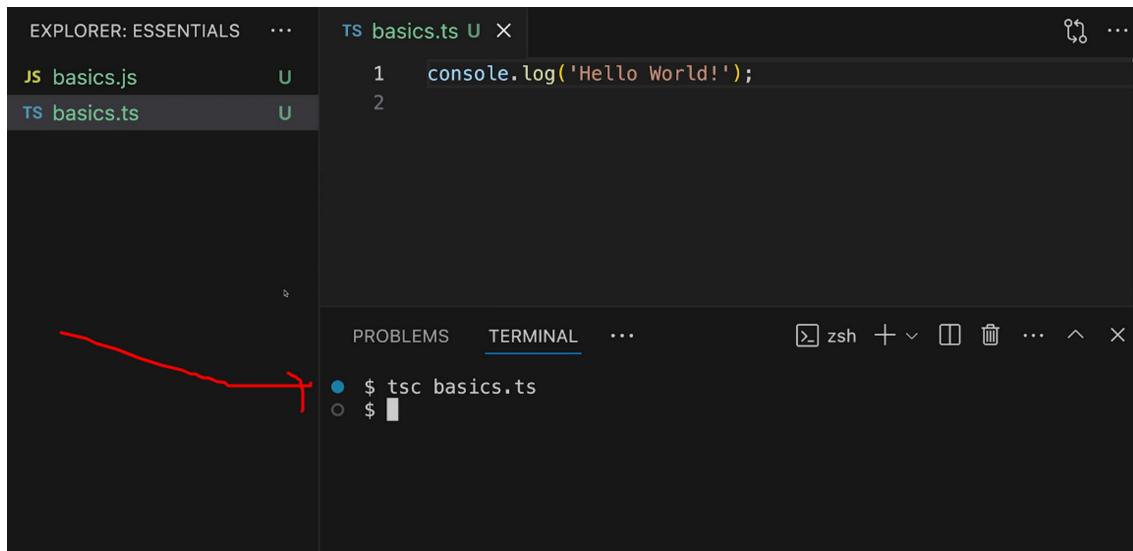
In this and many other sections, I'll occasionally run compiled code with Node.js - simply to save us the hassle of building a (demo) website that loads and executes that code.

You can install the Node.js runtime from <https://nodejs.org/en> (it's available for all operating systems). This runtime allows you to run JavaScript code right on your system (via the `node` command) instead of requiring a website.

So when you see me run `node app.js` (or some other file), that's me using Node.js to execute the code.

Project Setup

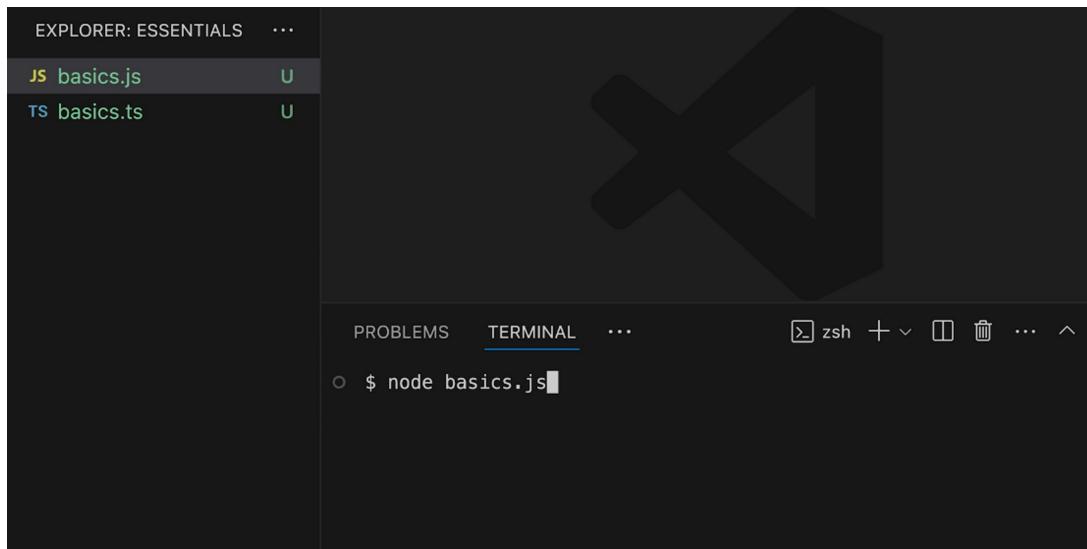
We are not going to build a website or anything like that. Instead we will write our typescript code in simple typescript files and we will execute them with node.js



A screenshot of the Visual Studio Code interface. On the left, the Explorer sidebar shows two files: 'basics.js' and 'basics.ts'. The 'basics.ts' file is currently selected. In the main editor area, the code 'console.log('Hello World!');' is written. Below the editor is the Terminal panel, which displays the command '\$ tsc basics.ts' followed by a cursor. A red arrow points from the text 'Now let's execute javascript file' to the terminal icon in the terminal bar.

Now let's execute javascript file

"node basics.js"



A screenshot of the Visual Studio Code interface. The Explorer sidebar shows 'basics.js' and 'basics.ts'. The 'basics.js' file is selected. The main editor area is dark, showing a large 'X' logo. The Terminal panel shows the command '\$ node basics.js' with a cursor. The status bar at the bottom indicates the file is being processed.

Working with Types & Exploring Built-in Types



A screenshot of the Visual Studio Code interface. The Explorer sidebar shows 'basics.ts'. The main editor area contains the following TypeScript code:

```
1 let userName: string;
2
3 // ...
4
5 userName = 'Max';
6
```

A yellow tooltip with a lightbulb icon appears over the word 'string' in the first line of code, indicating a code completion or type inference suggestion.

Vanilla JavaScript has Types too!

```
TS basics.ts X
1 let userName: string;
2
3 // ...
4
5 userName = 'Max';
6
```

💡 What's being said?

You're talking about **type assignments** in **TypeScript**, like this:

ts

```
let myName: string = "Khalid";
```

🧠 Key Points:

✓ 1. Type assignments like `: string`, `: number`, `: boolean` are TypeScript-specific

In JavaScript, you don't do this kind of stuff. TypeScript lets you **declare types** for variables, which helps with:

- Code safety 🚨
- Better IntelliSense in editors 💻
- Fewer bugs 😬

✗ 2. "string" should be lowercase

ts

```
let myName: string; ✓
```

```
let myName: String; ✗
```

Why? Because `string` is a **primitive type**, while `String` (uppercase) refers to a class — not what you want 99.9% of the time.

So always use lowercase for:

- `string`
- `number`
- `boolean`
- `object`
- `undefined`
- `null`

⌚ 3. JS vs TS — What's the difference?

Feature	JavaScript	TypeScript
---------	------------	------------

Type assignment (<code>: string</code>)	✗	✓
---	---	---

Runtime value types	✓	✓
---------------------	---	---

Compile-time safety



Even JavaScript understands **value types**, like when you do:

```
js
const age = 25;
console.log(typeof age); // "number"
```

But JS doesn't let you **enforce** that age must be a number — that's where TypeScript comes in and slaps on a type rule 😅

💡 In Summary:

- : string, : number = TypeScript magic 🌟
- Use lowercase (string, not String) — always
- JavaScript knows value types too, but doesn't **enforce** them like TypeScript

The screenshot shows the VS Code interface with two files open: `basics.ts` and `vanilla-js-types.js`.

basics.ts:

```
1  let userName = 'Max';
2  console.log(typeof userName);
```

vanilla-js-types.js:

```
1  let userName = 'Max';
2  console.log(typeof userName);
```

In the `basics.ts` file, the `typeof` operator is highlighted in yellow, indicating a TypeScript-specific feature.

PROBLEMS (1)

- \$ node vanilla-js-types.js
 string
 ○ \$ █

Type Inference vs Type Assignment

TS basics.ts X

```
1 let userName: string;
2
3 // ...
4
5 userName = 'Max';
6
```

A red callout points from the word "string" in the first line of code to the text "Type Assignment" located at the bottom right of the editor window.

TS basics.ts ●

```
1 let userName: string;
2 let userAge = 38;
3
4 // ...
5
6 userName = 'Max';
7
```

A red callout points from the variable declaration "userAge" in line 2 to the text "TypeScript is able to infer the type" located at the bottom right of the editor window.

Assigning Types to Function Parameters

```
ts basics.ts 2, M X ...
1 let userName: string; // number, boolean
2 let userAge = 38;
3
4 // ...
5
6 userName = 'Max';
7 // userAge = '34';
8
9 function add(a: number, b = 5) {
10   return a + b;
11 }
12
13 add(10);
14 add('10');
15 add(10, 6);
16 add(10, '6');
17
18
```

The "any" type

We always worked with exactly one type, one kind of value. Username must be string, user age must be a number.

But you sometimes have situations where you need more flexibility and that's why typescript also offers more flexible types.

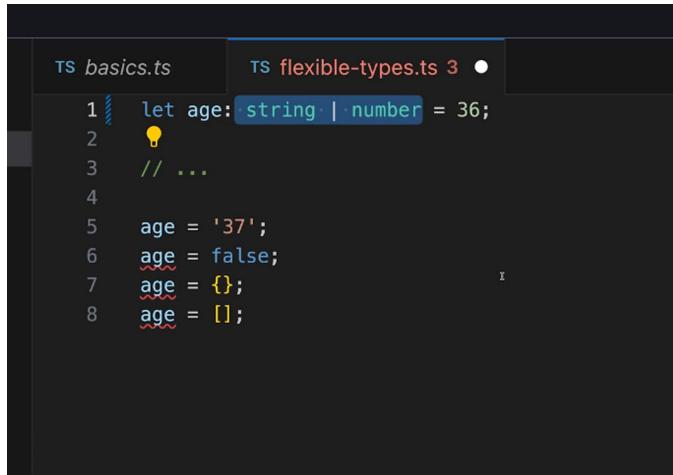
With 'any' we are allowing anything.

```
EXPLORER: ESSENTIALS ... TS basics.ts TS flexible-types.ts U ●
TS basics.ts
TS flexible-types.ts U
JS vanilla-js-types.js
```

```
1 let age: any = 36;
2
3 // ...
4
5 age = '37';
6 age = false;
7 age = {};
8 age = [];
```

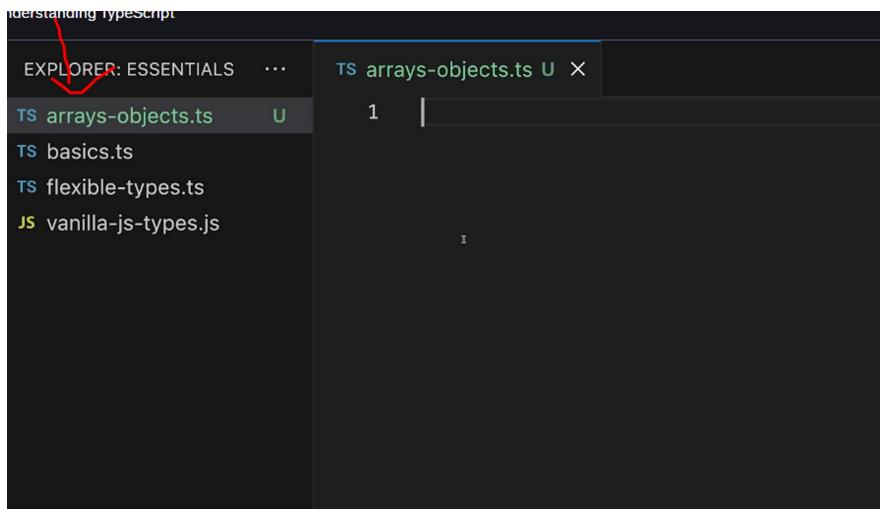
Understanding Union Types

With any type we are basically allowing anything which defeats the purpose of typescript.



```
TS basics.ts          TS flexible-types.ts 3 ●
1 let age: string | number = 36;
2
3 // ...
4
5 age = '37';
6 age = false;
7 age = {};
8 age = [];
```

Arrays & Types



understanding-typescript

EXPLORER: ESSENTIALS ...

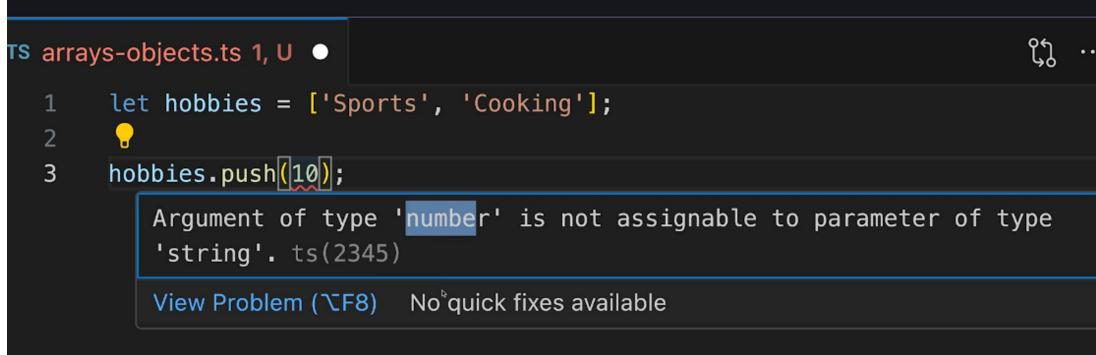
TS arrays-objects.ts U

TS basics.ts

TS flexible-types.ts

JS vanilla-js-types.js

This shows how powerful typescript is!



```
TS arrays-objects.ts 1, U ●
```

```
1 let hobbies = ['Sports', 'Cooking'];
2
3 hobbies.push([10]);
```

Argument of type 'number' is not assignable to parameter of type 'string'. ts(2345)

View Problem (F8) No quick fixes available

Advanced Array Types

```
TS arrays-objects.ts U ●
1  let hobbies = ['Sports', 'Cooking'];
2
3  // hobbies.push(10);
4  ⚡
5  let users: string[];
```

```
TS arrays-objects.ts U ●
1  let hobbies = ['Sports', 'Cooking'];
2
3  // hobbies.push(10);
4  ⚡
5  let users: (string | number)[];
6
7  users = [1, 'Max'];
8  users = [5, 1];
9  users = ['Max', 'Anna'];
```

A First Glimpse At Generic Types - Alternative Array Type Declaration

```
TS arrays-objects.ts U X
1  let hobbies = ['Sports', 'Cooking'];
2
3  // hobbies.push(10);
4  ⚡
5  let users: (string | number)[];
```

I wanna show you alternative way of writing this

```
1 let hobbies = ['Sports', 'Cooking'];
2
3 // hobbies.push(10);
4
5 // let users: (string | number) [];
6 let users: Array<string | number>;  
7
8 users = [1, 'Max'];
9 users = [5, 1];
10 users = ['Max', 'Anna'];
```



This is so called generic type. You could also use this. We will learn more on generic later in this course.

Generic type => combination of multiple type.

Making Sense of Tuples

There is also another kind of array type.

There is so called tuple type.

```
s arrays-objects.ts ●
```

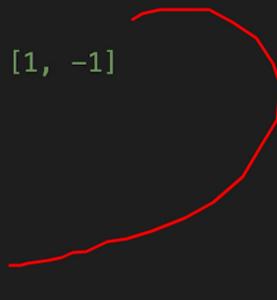
```
6 let users: Array<string | number>;
7
8 users = [1, 'Max'];
9 users = [5, 1];
10 users = ['Max', 'Anna'];
11
12 |
```

Let's assume we want to just store two values in a array. Not more other than that.

So, we could define the array like this

```
arrays-objects.ts ●
```

```
8 users = [1, 'Max'];
9 users = [5, 1];
10 users = ['Max', 'Anna'];
11
12 let possibleResults: number[]; // [1, -1]
13
14 possibleResults = [1, -1];
15 possibleResults = [5, 10, 12];
```



You might wonder why would I do that? But imagine you are working in a big team and not all colleagues are aware of it!

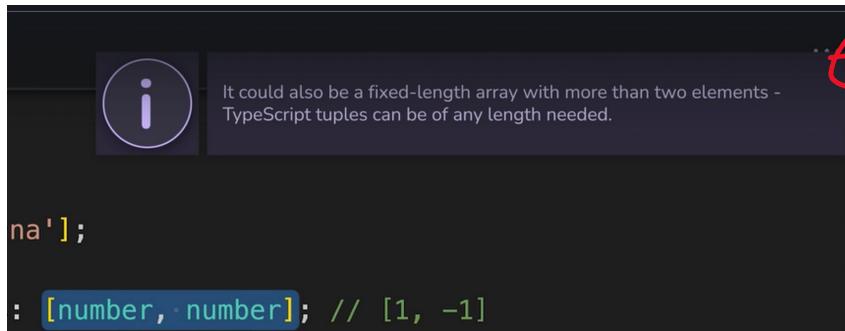
That's where 'tuple' comes into place.

```
TS arrays-objects.ts 1 ●
```

```
7
8 users = [1, 'Max'];
9 users = [5, 1];
10
```

```
8  users = [1, 'Max'];
9  users = [5, 1];
10 users = ['Max', 'Anna'];
11
12 let possibleResults: [number, number]; // [1, -1]
13
14 possibleResults = [1, -1]; Accepted
15 possibleResults = [5, 10, 12]; Rejected
```

By doing it, you are telling typescript that possible results is a tuple with exactly two values, where the first value will be of type number which is why this value here is accepted. Second value is rejected



Object Types

Inferred Object

```
TS arrays-objects.ts ●
14  possibleResults = [1, -1];
15  // possibleResults = [5, 10, 12]
16
17 let user = {
18   name: 'Max',
19   age: 38
20 };
```

TS arrays-objects.ts ●

```
14  possibleResults = [1, -1];
15  // possibleResults = [5, 10, 12];
16
17  let user = {
18    na
19    ag
20  };
    let user: {
      name: string;
      age: number;
    }
```

TS arrays-objects.ts 1 ● ...

```
14  possibleResults = [1, -1];
15  // possibleResults = [5, 10, 12];
16
17  let user: {
18    name: string;
19    age: number | string;
20    hobbies: string[];
21    role: {
22      description: string;
23      id: number;
24    }
25  } = {
26    name: 'Max',
27    age: 38
28 };
```

```
let user: {
  name: string;
  age: number | string;
  hobbies: string[];
  role: {
    description: string;
    id: number;
  }
} = {
  name: 'Max',
  age: 38
};
```

```
TS arrays-objects.ts ●
17 let user: {
18   name: string;
19   age: number | string;
20   hobbies: string[];
21   role: {
22     description: string;
23     id: number;
24   }
25 } = {
26   name: 'Max',
27   age: 38,
28   hobbies: ['Sports', 'Cooking'],
29   role: [
30     {
31       description: 'admin',
32       id: 5
33     }
34   ];
}
```

Tricky: The "Must Not Be Null" type

✳️ The Confusing Part: {} as a Type

💡 Example:

ts

CopyEdit

let val: {} = "hello";

You'd think this should throw an error, right? Like...

“Bro, val is supposed to be an object, how tf is a string okay?”

But TypeScript just shrugs and says:

"Cool, whatever bro." 😊

💀 Why This Happens?

Because in **TypeScript**, when you use `{}` as a type, it actually **does NOT mean "empty object"**.

! It actually means:

"any non-null, non-undefined value"

So:

```
ts
CopyEdit
let val: {} = "hello";      // ✅ allowed
let val: {} = 123;          // ✅ allowed
let val: {} = false;        // ✅ allowed
let val: {} = [];           // ✅ allowed
let val: {} = {};
```

// ✅ allowed

```
let val: {} = null;        // ❌ error
let val: {} = undefined;    // ❌ error
```

💡 What {} does NOT mean:

It does **not** mean:

```
ts
CopyEdit
{
  // no properties allowed
}
```

That's what most people (including us devs from a C-like background) **expect**, but TypeScript ain't doing that here 😱

💡 So What Should You Use Instead?

If you *actually* want to allow only real object types (like objects with keys/props), you should use:

```
ts
CopyEdit
let val: object; // ✅ only accepts object types, not primitives like string/number
Or if you want a real empty object type:
```

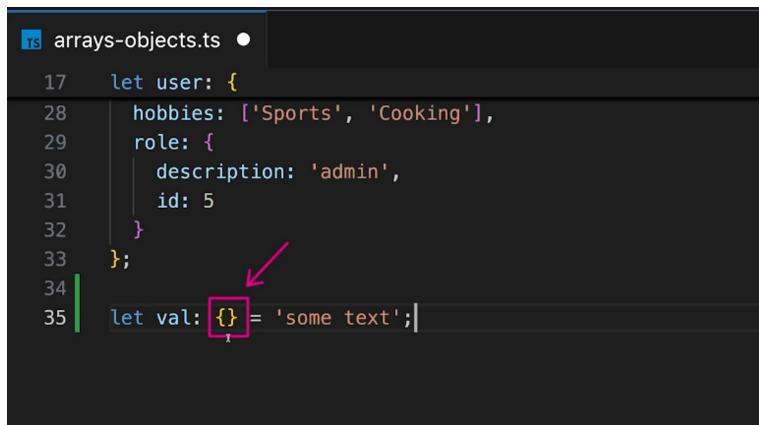
```
ts
CopyEdit
let val: { [key: string]: any }; // ✅ allows any key-value pairs
let val: Record<string, any>; // ✅ same thing, cleaner syntax
```

⚠ Summary (Sticky Notes Style):

Type	What it really means	Allows
<code>{}</code>	Anything except null or undefined	string, number, boolean, array, object ✅
<code>object</code>	Real JS object types only	<code>{}, [], { name: "Khalid" }</code> ✅ <code>"hello", 123</code> ❌
<code>Record<string, any></code>	Key-value object with string keys	<code>{ name: "Khalid" }</code> ✅

💡 Real Talk Tip:

If you're ever confused by TypeScript types — run them through the TS Playground and see what happens. That thing is a blessing 😊

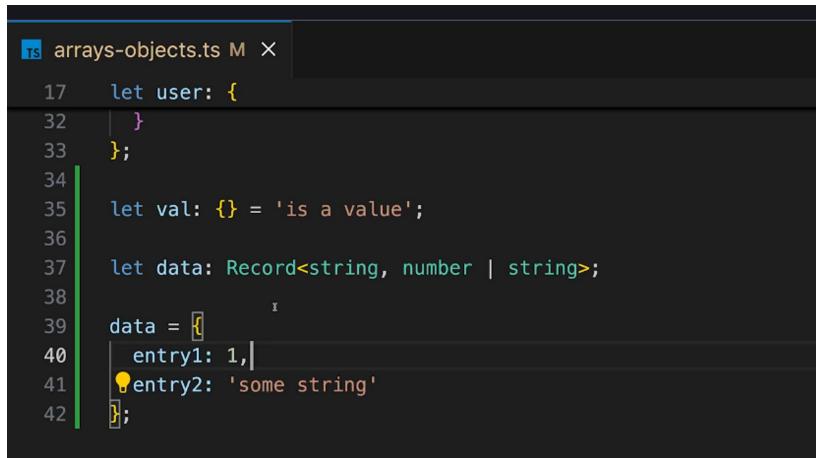


```
ts arrays-objects.ts ●
17 let user: {
18   hobbies: ['Sports', 'Cooking'],
19   role: {
20     description: 'admin',
21     id: 5
22   }
23 };
24 let val: {} = 'some text';
```



```
ts arrays-objects.ts 1 ●
17 let user: {
18   hobbies: ['Sports', 'Cooking'],
19   role: {
20     description: 'admin'.
21     Type 'null' is not assignable to type '{}'. ts(2322)
22   }
23 };
24 let val: {} = null;
25 View Problem (F8) No quick fixes available
```

Flexible Objects with the Record Type



```
ts arrays-objects.ts M X
17 let user: {
18   hobbies: ['Sports', 'Cooking'],
19   role: {
20     description: 'admin',
21     id: 5
22   }
23 };
24 let val: {} = 'is a value';
25 let data: Record<string, number | string>;
26 data = {
27   entry1: 1,
28   entry2: 'some string'
29 };
```

⌚ The Scenario

You wanna make a variable (data) that holds an object with **unknown keys**, like:

ts

```
const data = {  
  name: "Khalid",  
  age: 25,  
  email: "khalid@email.com",  
};
```

But here's the twist:

You don't know yet which keys will be in the object — just that it will be *some object*, and the keys will be strings, and the values will be of a certain type.

✗ Why this won't work:

ts

```
let data: {} = {};
```

This just means: "**any value that's not null or undefined**"

So it's *too loose*, and doesn't actually restrict data to be a proper object.

✓ Use Record<KeyType, ValueType>

Example:

ts

```
let data: Record<string, string | number> = {};
```

This says:

- All keys must be **strings** 🔑
- All values must be **string or number** 💬 12 34

Now you can do:

ts

```
data["name"] = "Khalid"; // ✓  
data["age"] = 25; // ✓  
data["email"] = "yo@abc.com"; // ✓  
data["isActive"] = true; // ✗ Error: 'boolean' not allowed
```

🎯 Real-life Use Case (Alumni Website 🎓)

Let's say you're building a students object:

ts

```
const students: Record<string, { name: string, gradYear: number }> = {};
```

Then you can dynamically add students:

ts

```
students["student001"] = { name: "Khalid", gradYear: 2020 };  
students["student002"] = { name: "Rafi", gradYear: 2022 };
```

Boom! You've got type safety *and* flexibility. 😊

Behind the Scenes (Generics)

When you see this:

ts

CopyEdit

Record<string, number>

You're using a **generic type**. Record is a generic, meaning it's like a function for types — you plug in:

- What kind of keys you want (string, number)
- What kind of values you want (string, boolean, object, whatever)

And it builds the type *on the fly*.

TL;DR Summary

Concept

What it means

{ } as type Any value except null/undefined (not ideal)

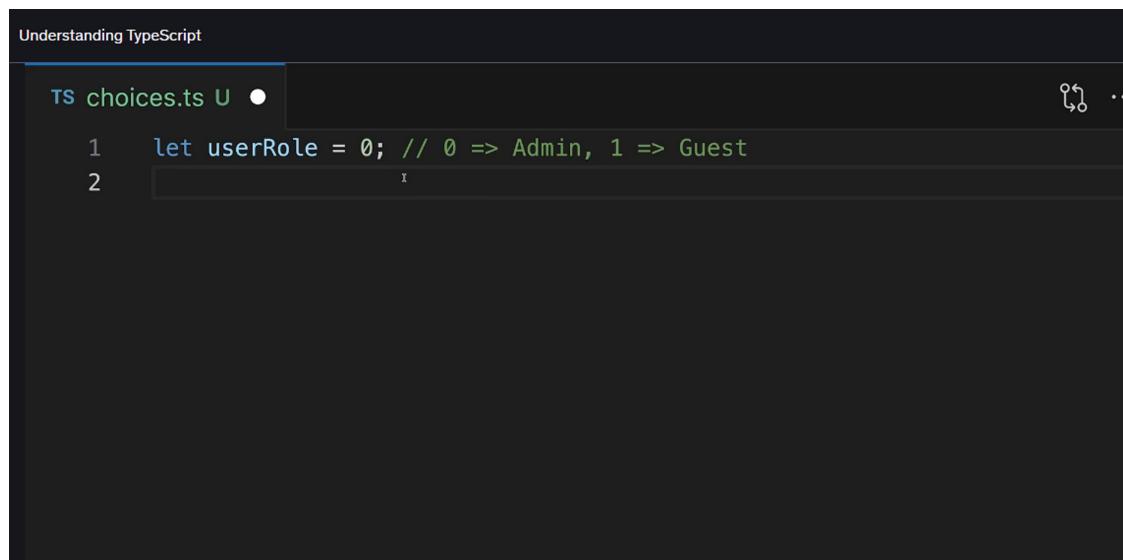
object as type Only object values, no primitives

Record<K, V> Keys of type K, values of type V

Record<string, any> Object with unknown string keys & any values

If you're building anything dynamic or key-based (like a dictionary, a settings object, a student registry, etc), **Record is your best friend** 

Working with Enums



The screenshot shows a code editor window titled "Understanding TypeScript". The file tab shows "TS choices.ts U". The code editor displays the following TypeScript code:

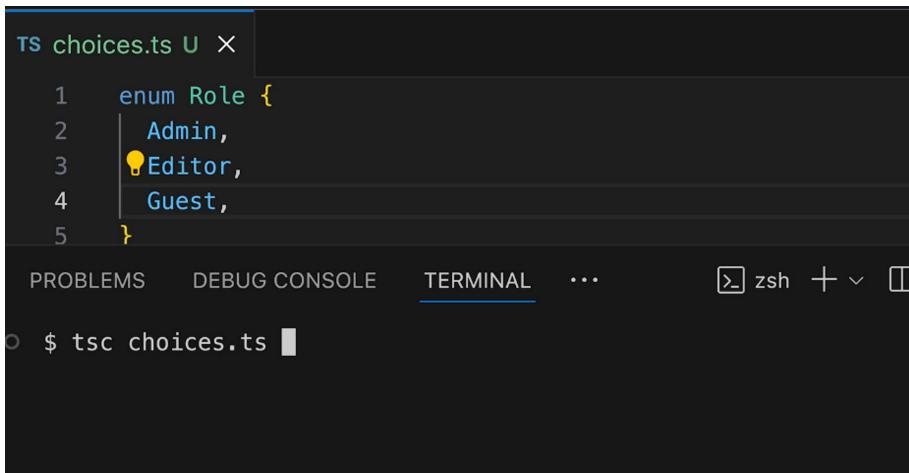
```
1 let userRole = 0; // 0 => Admin, 1 => Guest
```

```
TS choices.ts U ●
1 enum Role {
2   Admin,
3   Editor,
4   Guest,
5 }
6
7 let UserRole = 0;
8
```

```
TS choices.ts 1, U ●
1 enum Role {
2   Admin,
3   Editor,
4   Guest,
5 }
6
7 let UserRole: Role = 5;
8
```

```
TS choices.ts U ●
1 enum Role {
2   Admin,
3   Editor,
4   Guest,
5 }
6
7 let UserRole: Role = Role.Admin;
8
9 // ...
10
11 UserRole = Role.Guest;
12
```

If I compile this .ts file



```
ts choices.ts U X
1 enum Role {
2   Admin,
3   Editor,
4   Guest,
5 }
```

PROBLEMS DEBUG CONSOLE TERMINAL ... zsh + ~

```
○ $ tsc choices.ts
```

This is our compiled js code



```
js choices.js U X
1 var Role;
2 (function (Role) {
3   Role[Role["Admin"] = 0] = "Admin";
4   Role[Role["Editor"] = 1] = "Editor";
5   Role[Role["Guest"] = 2] = "Guest";
6 })(Role || (Role = {}));
7 var userRole = Role.Admin;
8 // ...
9 userRole = Role.Guest;
10
```

You can also store string



```
ts choices.ts 1, U ●
1 enum Role {
2   Admin = 'Admin',
3   Editor = 'Editor',
4   Guest = 'Guest',
5 }
```

📌 The Problem

You're tracking **user roles** like this:

ts

CopyEdit

```
let userRole = 0; // 0 = admin, 1 = editor, 2 = guest
```

🧠 Sure, it works... but what the heck does 0 even mean without a comment? It's not readable, not safe, and *low-key* messy.

The Solution: enum

 Declare it like a boss:

ts

```
CopyEdit
enum Role {
  Admin,
  Editor,
  Guest,
}
```

Boom. Role is now a custom type with only **3 allowed values**: Role.Admin, Role.Editor, and Role.Guest.

 Use it like this:

ts

```
CopyEdit
let userRole: Role;
userRole = Role.Admin; // ✅
userRole = Role.Guest; // ✅
userRole = 0;          // ✅ also works, because Admin = 0 internally
userRole = 5;          // ❌ not allowed, TypeScript will complain
So internally:
```

ts

```
CopyEdit
Role.Admin = 0
Role.Editor = 1
Role.Guest = 2
```

But *externally*, your code is now **self-documenting** and  type-safe.

Make it even clearer (Custom Values!)

Let's say you want to **manually assign values**, like strings:

ts

```
CopyEdit
enum Role {
  Admin = "ADMIN",
  Editor = "EDITOR",
  Guest = "GUEST",
}
```

This is **100%** for APIs, databases, or configs where you care about **exact values**.

ts

```
CopyEdit
let userRole: Role = Role.Editor;
if (userRole === Role.Editor) {
  console.log("Welcome, editor!");
}
```

TL;DR Cheat Sheet

Feature	What it does
---------	--------------

enum Role { ... } Creates a new type with specific allowed values
Default enum values Are numbers starting from 0
Custom values You can use strings (or numbers) manually
Safer than constants Prevents invalid assignments
Improves readability Makes your code hella clean and descriptive 

Quick Example in a React Context:

```
ts
CopyEdit
enum Theme {
  Light = "light",
  Dark = "dark",
}
const [theme, setTheme] = useState<Theme>(Theme.Light);
const toggleTheme = () => {
  setTheme(prev => prev === Theme.Light ? Theme.Dark : Theme.Light);
};
Super clean, type-safe, and  avoids accidental string mistakes.
```

Being Specific With Literal Types

So this enum feature can be really useful if you have a set number of choices you want to use in your code. There also is an alternative to using an enum though, which arguably is a bit more popular. So you might find enums in certain code bases, but there also are developers who try to avoid them. In the end, it comes down to personal preferences, but it's important to know about the arguably more popular alternative to enums. So instead of defining such a role enum here, instead of using that,



```
ts choices.ts U X
1  enum Role {
2    Admin,
3    Editor,
4    Guest,
5  }
6
7  let userRole: Role = 0;
8
9  // ...
10
11 userRole = Role.Guest;
12
```

you could also use a union type here
for the `userRole` variable

combined with another TypeScript feature
you haven't learned about yet: literal types.
Because in TypeScript you cannot just set up

general types like number or string, you could say,
but you can also define very specific values as types.
For example, you could set the string `admin`
as a type for `userRole`.

```
TS choices.ts 2, U ●
1  // enum Role {
2  //   Admin,
3  //   Editor,
4  //   Guest,
5  // }
6  let userRole: 'admin' = 0;
7
8
9  // ...
10
11  userRole = Role.Guest;
12
```

```
TS choices.ts 1, U ●
1  // enum Role {
2  //   Admin,
3  //   Editor,
4  //   Guest,
5  // }
6  let userRole: 'admin' = 'admin';
7
8
9  // ...
10
11  userRole = Role.Guest;
12
```

Now this might look very weird
because this very much looks like a string value,
and it would be if it were on the right side
of the equal sign.

But here on the left side of the equal sign after this colon, no matter if you're using it on a variable or a parameter, in a function or anywhere else, this thing here is actually not a value but a type.

And it tells TypeScript that the allowed value for `userRole` is this very specific string.



```
TS choices.ts 1, U ●

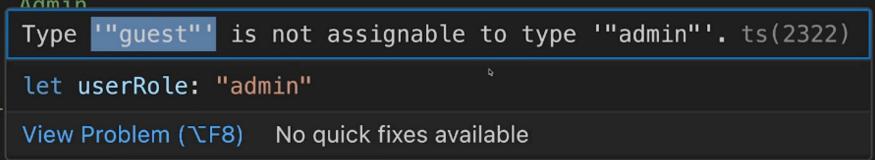
1  // enum Role {
2  //   Admin,
3  //   Editor,
4  //   Guest,
5  // }
6
7 let UserRole: 'admin' = 'admin';
8
9 // ...
10
11 UserRole = Role.Guest;
12
```

So if I were to store the string `guest` in there, I would get an error that the type `guest` is not assignable to this very specific string type.

Now you might wonder what the use of this very specific string type is. After all, this does not give us any flexibility at all.

Well, it gets more useful if you turn it into a union type

and you tell TypeScript that `userRole` should either be the string `admin` or the string `editor`, let's say, or the string `guest`. Now we can initialize this to `admin`, but later on, instead of using an enum,



```
TS choices.ts 2, U ●

1  // enum Role {
2  //   Admin
3  //   Type '"guest"' is not assignable to type '"admin"'. ts(2322)
4  //   Editor
5  //   Guest
6  // }
7  let UserRole: 'admin' = 'guest';
8
9 // ...
10
11 UserRole = Role.Guest;
12
```

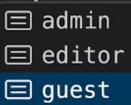
```
TS choices.ts 1, U ●
```

```
1 // enum Role {  
2 //   Admin,  
3 //   Editor,  
4 //   Guest,  
5 // }  
6    
7 let UserRole: 'admin' | 'editor' | 'guest' = 'admin';  
8  
9 // ...  
10  
11 UserRole = Role.Guest;  
12
```

And I'm even getting auto-completion in my IDE here because it understands which options we have. And that's another very convenient way of working with options.

And as mentioned, it's arguably more popular in the TypeScript community than using an enum. But you can use both alternatives

```
TS choices.ts 1, U ●
```

```
1 // enum Role {  
2 //   Admin,  
3 //   Editor,  
4 //   Guest,  
5 // }  
6  
7 let UserRole: 'admin' | 'editor' | 'guest' = 'admin';  
8  
9 // ...  
10    
11 UserRole = '';  
12 

- └─ admin
- └─ editor
- └─ guest

```