

# **TYPESCRIPT**

## Step 1 - 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 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 ✅

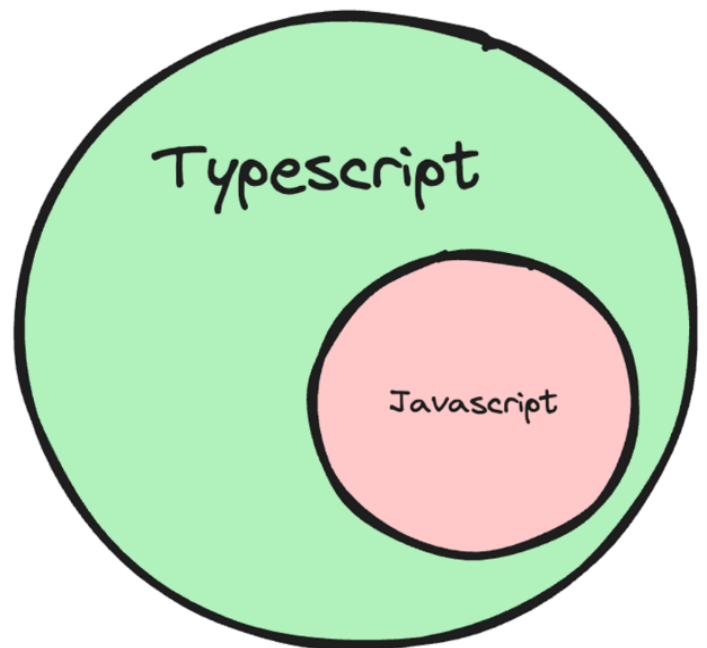
```
function main() {
    let number = 10;
    number = "text";
    return number;
}
```

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.

## What is typescript?

TypeScript is a programming language developed and maintained by Microsoft.

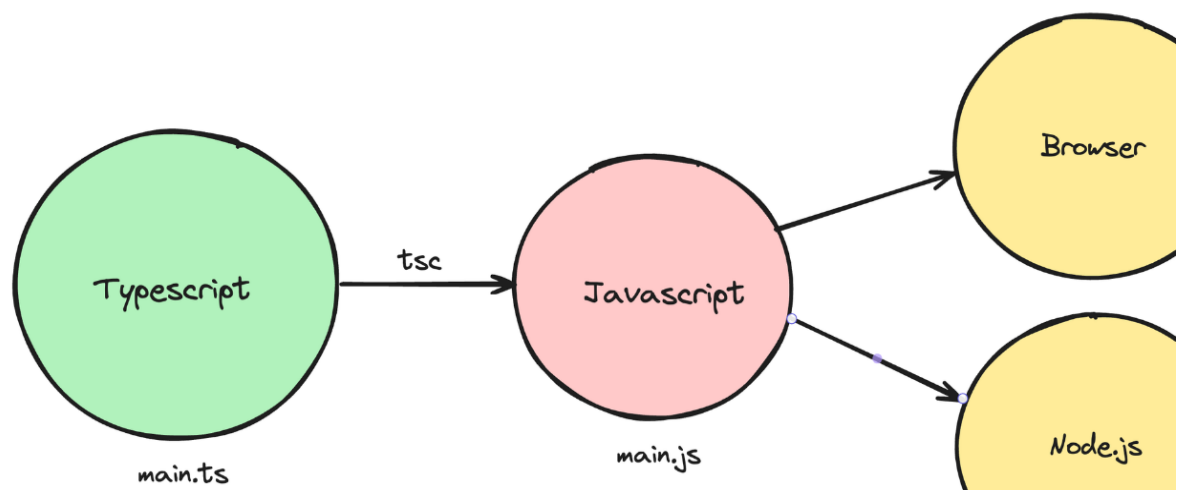
It is a strict **syntactical superset** of JavaScript and adds optional static typing to the language.



### 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 is compiled down to javascript.

Browser can't understand typescript.

## 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

Steps to make the tsc compiler so that it could work on vs code:-

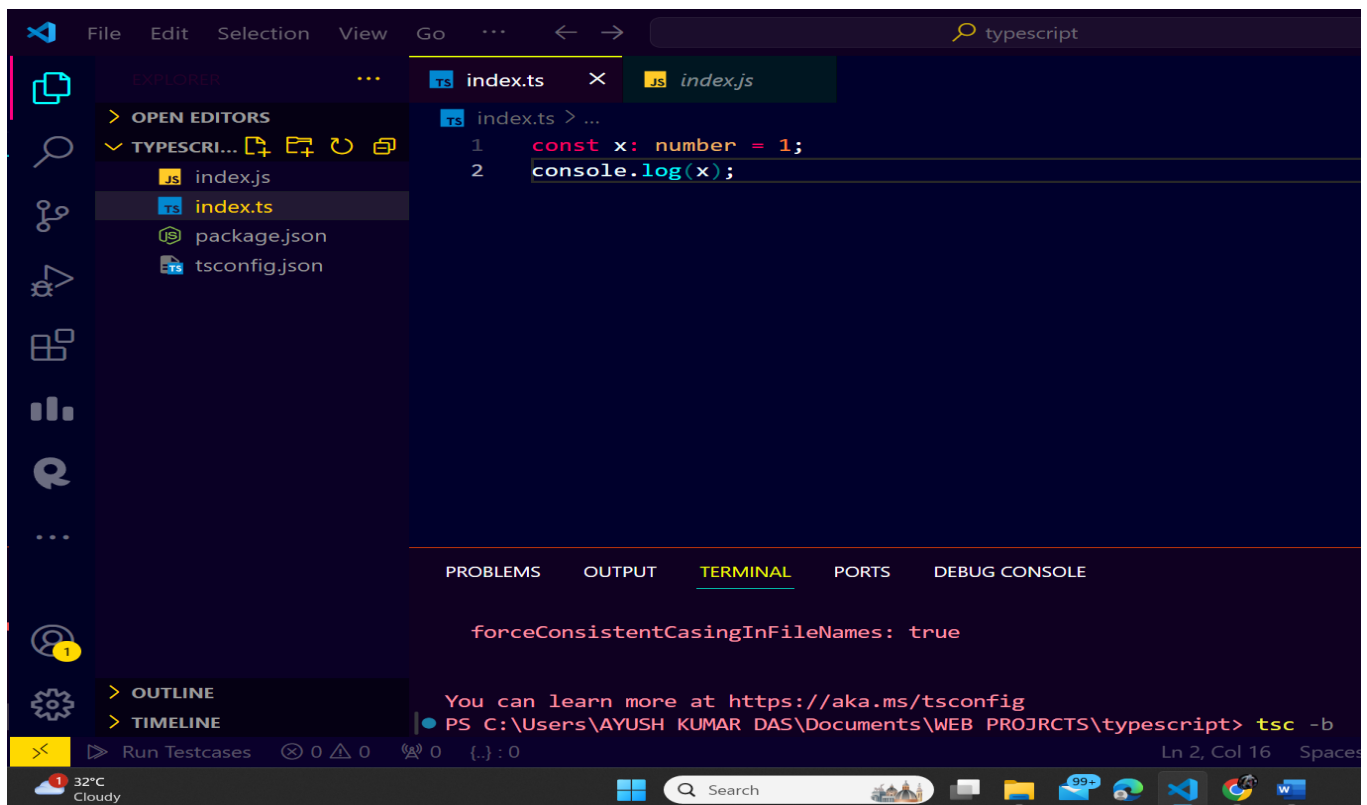
Step 1:- Create a file name with the extension of (.tsc).

Step 2:- Now open the terminal and write this in the following order:-

```
npm init -y  
npx tsc --init  
tsc -b
```

tsc -b will run your (.ts) file.

now you will see this if everything goes right:-



Index.js, package.json, and tsconfig.json are newly generated.

So whatever you write in index.ts file then after compilation, the code of (.ts) file will be compiled down to (.js) file.

How your index.ts file will be compiled down to an index.js file will depend upon how we configure tsconfig.json.

The final file that will run is the index.js file.

```
s a.ts > ...  
1   let x: number = 101;  
2   x = "harkirat";  
3   console.log(x);  
4  
5
```

This will provide an error in typescript but not in javascript.

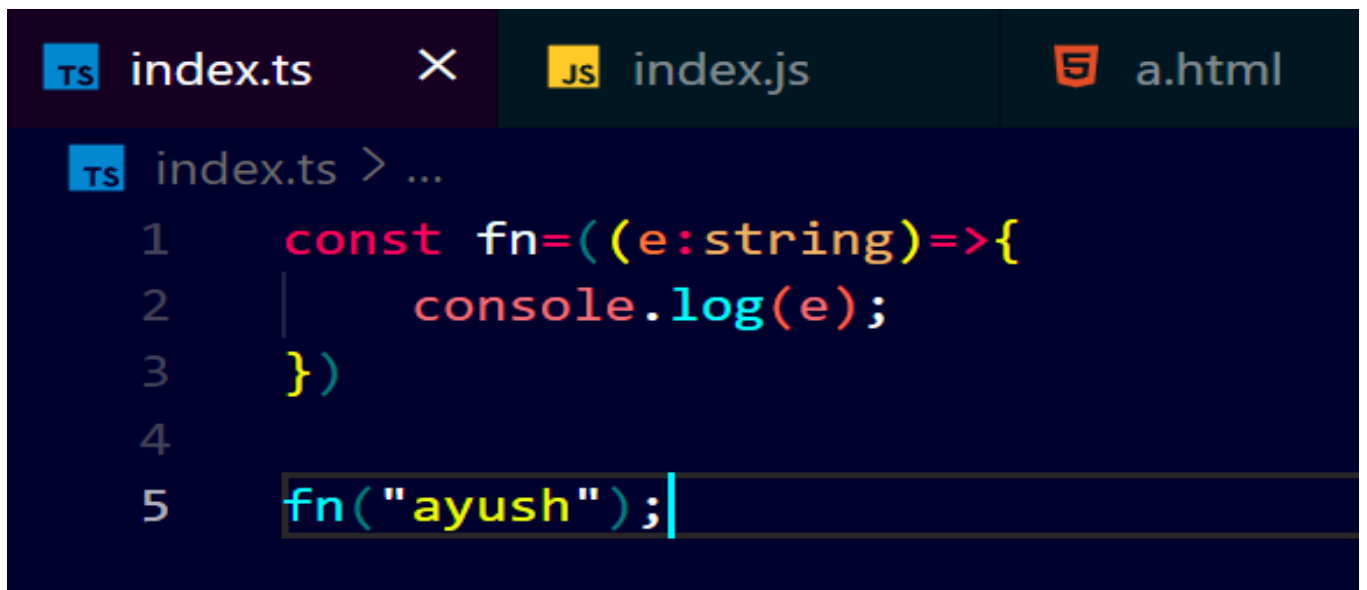
Some basic types in typescript:-

## Typescript provides you some basic types

`number`, `string`, `boolean`, `null`, `undefined`.

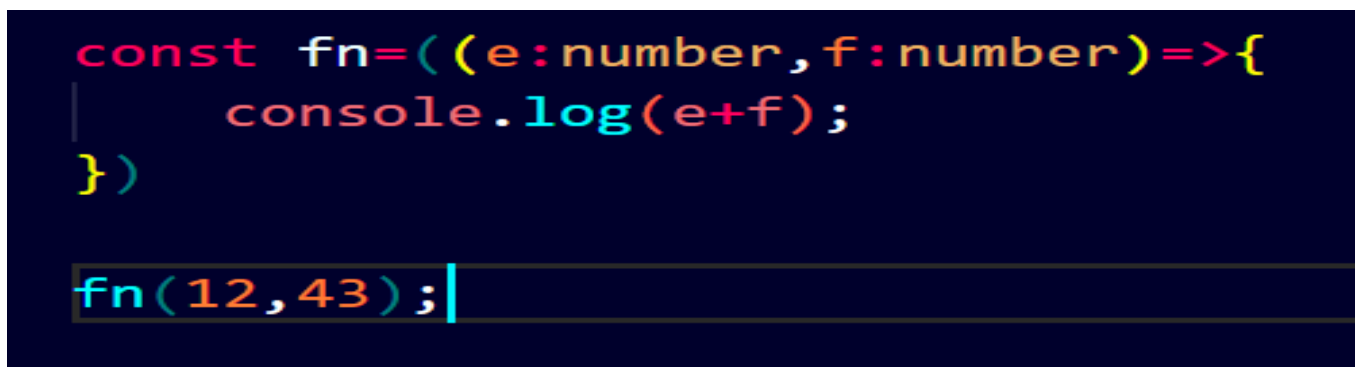
Let's create some simple applications using these types -

1) Let's create a simple function to print name:-



```
TS index.ts × JS index.js a.html
TS index.ts > ...
1  const fn=((e:string)=>{
2    |    console.log(e);
3  })
4
5  fn("ayush");
```

2) Write a function for summation of two no.s:-



```
const fn=((e:number,f:number)=>{
|    console.log(e+f);
})

fn(12,43);
```

The difference is we have to provide the datatype of variables in a typescript file like C++, whereas we don't have to provide the datatype in the javascript file.

3) Return true if the person in above 18:-

```
const fn=((e:number):string=>{  
  ⚡ if(e>=18){  
    return "aaja bhai";  
  }  
  else{  
    return "bhag";  
  }  
})  
let k:string=fn(12);  
console.log(k);
```

The e:number tells what type of datatype is sending to the function and (:string) tells the datatype of the string that is returned by the function.

- 4) Write a function that another function as an input and run after 1 sec:-

```
const fn=((gn:()=>void)=>{  
  setTimeout(gn,1000);  
})  
fn(function(){  
  console.log("ues");  
})
```

gn:()=>void tells that the gn function is not returning anything.

## THE tsconfig.json 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.

To try it out, try compiling the following code for target being `ES5` and `es2020`

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

► Output for ES5

► Output for ES2020

Based on the ECMAScript version typescript compile the code.



## 2. rootDir

Where should the compiler look for `.ts` files. Good practise is for this to be the `src` folder

## 3. outDir

Where should the compiler look for spit out the `.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

## 5. removeComments

Whether or not to include comments in the final `.js` file

For better practice create a folder name `src` where your `index.ts` file will get stored and also create a folder named as `dist` which will store the final output file of `index.js`.

Now in `tsconfig.json` give the new path of `index.ts` (root directory) and `index.js` (out directory).

```
2  "compilerOptions": {
24  // "useDefineForClassFields": true,
    ECMAScript-standard-compliant class fields. */
25  // "moduleDetection": "auto",
    to detect module-format JS files. */
26
27  /* Modules */
28  "module": "commonjs",
    generated. */
29  "rootDir": "./src",
    within your source files. */
30  // "moduleResolution": "node10",
    up a file from a given module specifier. */
31  // "baseUrl": "./",
    resolve non-relative module names. */
32  // "paths": {},
```

Now for compilation you have to do this:-

```
PS C:\Users\AYUSH KUMAR DAS\Documents\WEB PROJRCTS\typescript> node dist/index.js
```

Now in the tsconfig.json file make noImplicitAny to false.

```
"compilerOptions": {  
    // 'allowSyntheticDefaultImports' for type compatibility  
    // "preserveSymlinks": true,  
    // their realpath. This correlates to the same flag in  
    "forceConsistentCasingInFileNames": true,  
    // in imports. */  
  
    /* Type Checking */  
    "strict": true,  
    // type-checking options. */  
    "noImplicitAny": false,  
    // expressions and declarations with an implied 'any'  
    // "strictNullChecks": true,  
    // account 'null' and 'undefined'. */  
    // "strictFunctionTypes": true,  
    // check to ensure parameters and the return values are
```

But finally false, should be there.

**Interfaces:-**

# 1. What are interfaces

How can you assign types to objects? For example, a user object that looks 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 returns true or false if a user is above 18. It takes a user as an input.

▼ Solution

```
interface User {  
  firstName: string;  
  lastName: string;  
  email: string;  
  age: number;  
}  
  
function isLegal(user: User) {  
  if (user.age > 18) {  
    return true  
  } else {  
    return false;  
  }  
}
```

## 2. 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.

```
class Employee implements Person {  
  name: string;  
  age: number;  
  
  constructor(n: string, a: number) {  
    this.name = n;  
    this.age = a;  
  }  
  
  greet(phrase: string) {  
    console.log(`${phrase} ${this.name}`);  
  }  
}
```

This is useful since now you can create multiple **variants** of a person (Manager, CEO ...)

**Types:-**

# What are types?

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

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

But they let you do a few other things.

## 1. Unions

Let's say you want to print the `id` of a user, which can be a number or a string.



You can not do this using `interfaces`

```
type StringOrNumber = string | number;  
  
function printId(id: StringOrNumber) {  
  console.log(`ID: ${id}`);  
}  
  
printId(101); // ID: 101  
printId("202"); // ID: 202
```

## 2. Intersection

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

💡 You can not do this using `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"  
};
```

Types can't be used as classes while , interface  
can be used as classes.

## **Array:-**

If you want to access arrays in 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 in the array

▼ Solution

```
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]));
```

## **Enums:-**

Enums (short for enumerations) in TypeScript are a feature that allows 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 whether the user has pressed the `up` arrow key,

```
function doSomething(keyPressed) {  
    // 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`) ?

The best thing to use in such a case is an `enum`.

```
enum Direction {  
    Up,  
    Down,  
    Left,  
    Right  
}  
  
function doSomething(keyPressed: Direction) {  
    // do something.  
}  
  
doSomething(Direction.Up)
```

By using enums we have created a cleaner way ,  
we can also achieve this by using types.

We have just written a constants whose datatype is not defined.

The values assigned to enum directions is just the index no.s like they are stored in array.

```

1
2 enum Direction {
3     ... Up, // 0
4     ... Down, // 1
5     ... Left, // 2
6     ... Right // 3
7 }
8

```

These are the default values , but we can also

### 3. How to change values?

```

enum Direction {
    Up = 1,
    Down, // becomes 2 by default
    Left, // becomes 3
    Right // becomes 4
}

function doSomething(keyPressed: Direction) {
    // do something.
}

doSomething(Direction.Down)

```

change these values:-

## 4. Can also be strings

```
enum Direction {  
    Up = "UP",  
    Down = "Down",  
    Left = "Left",  
    Right = 'Right'  
}  
  
function doSomething(keyPressed: Direction) {  
    // do something.  
}  
  
doSomething(Direction.Down)
```

## 5. 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({});  
})
```

Generics in TypeScript allow you to create reusable components and functions that work with a variety of data types while maintaining type safety. They provide a way to write code that is more flexible and adaptable to different scenarios by allowing you to define placeholders for types that will be specified when the code is used.

Here's a basic example of generics in TypeScript:

```
typescript ✂ Save to grepper 📋 Copy code

// Define a generic function that accepts an argument of type T and returns an array
function toArray<T>(arg: T): T[] {
    return [arg];
}

// Usage examples
let arr1 = toArray<number>(10); // Specify the type explicitly
let arr2 = toArray("hello"); // TypeScript infers the type automatically
```

## Generics:-

Don't read the above example.

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

Simple example -

► Code

JavaScript ▾

```
function identity<T>(arg: T): T {
    return arg;
}

let output1 = identity<string>("myString");
let output2 = identity<number>(100);
```

## **Importing and Exporting Modules:-**

TypeScript follows the ES6 module system, using `import` and `export`

### **1. Constant exports**

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));
```

### Constants Exports:

- Constants exports are similar to named exports but are used specifically for exporting constants.
- They are particularly useful when you want to export read-only variables that should not be modified after exporting.
- Example:


typescript

 Copy code

```
// Exporting a constant  
export const MAX_VALUE = 100;
```

- In another module, you can import the constant like this:

typescript

 Copy code


```
import { MAX_VALUE } from './module';  
console.log(MAX_VALUE); // Output: 100
```



## 1. Named Exports:

- With named exports, you can export multiple entities from a module by specifying their names.
- Each exported entity can be imported individually by its name in other modules.
- Example:


typescript

 Copy code

```
// Exporting multiple entities with named exports
export const PI = 3.14;
export function square(x: number): number {
    return x * x;
}
export class Circle {
    // Class implementation...
}
```

- In another module, you can import the named exports like this:

typescript


 Copy code

```
import { PI, square, Circle } from './module';
console.log(PI); // Output: 3.14
console.log(square(2)); // Output: 4
```

## Default Export:

- With default export, you can export a single entity (variable, function, class, etc.) as the default export of a module.
- There can only be one default export per module.
- Example:


typescript

 Copy code

```
// Exporting a single entity as the default export
const defaultValue = 'This is the default export';
export default defaultValue;
```

- In another module, you can import the default export like this:

typescript

 Copy code

```
import defaultValue from './module';
console.log(defaultValue); // Output: This is the default export
```

## **Pick :-**

# Pick

**Pick** allows you to create a new type by selecting a set of properties ( **Keys** ) from an existing type ( **Type** ).

Imagine you have a User model with several properties, but for a user profile display, you only need a subset of these properties.

```
interface User {
  id: number;
  name: string;
  email: string;
  createdAt: Date;
}

// For a profile display, only pick `name` and `email`
type UserProfile = Pick<User, 'name' | 'email'>;

const displayUserProfile = (user: UserProfile) => {
  console.log(`Name: ${user.name}, Email: ${user.email}`);
};
```

Now it is not necessary that all the things in UserProfile should be updated sometimes user just want to update certain things.

## **Partial :-**



# Partial

**Partial** makes all properties of a type optional, creating a type with the same properties, but each marked as optional.

```
interface User {  
  name: string;  
  email: string;  
  image: string;  
}
```



```
interface User {  
  name?: string;  
  email?: string;  
  image?: string;  
}
```

Specifically useful when you want to do **updates**

```
interface User {  
  id: string;  
  name: string;  
  age: string;  
  email: string;  
  password: string;  
};  
  
type UpdateProps = Pick<User, 'name' | 'age' | 'email'>  
  
type UpdatePropsOptional = Partial<UpdateProps>  
  
function updateUser(updatedProps: UpdatePropsOptional) {  
  // hit the database to update the user  
}  
  
updateUser({})
```

## **Readonly :-**

### **Readonly**

When you have a configuration object that should not be altered after initialization, making it **Readonly** ensures its properties cannot be changed.

```
interface Config {
  readonly endpoint: string;
  readonly apiKey: string;
}

const config: Readonly<Config> = {
  endpoint: 'https://api.example.com',
  apiKey: 'abcdef123456',
};

// config.apiKey = 'newkey'; // Error: Cannot assign to 'apiKey' because it is a read-only
```



This is compile time checking, not runtime (unlike const)

## **Record and Map :-**

Earlier way to type object is this:-

```
interface User {
  id: string;
  name: string;
}

type Users = { [key: string]: User };

const users: Users = {
  'abc123': { id: 'abc123', name: 'John Doe' },
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },
};
```

By using record we can do this:-

```
interface User {  
  id: string;  
  name: string;  
}  
  
type Users = Record<string, User>;  
  
const users: Users = {  
  'abc123': { id: 'abc123', name: 'John Doe' },  
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },  
};  
  
console.log(users['abc123']); // Output: { id: 'abc123', name: 'John Doe' }
```

By using map we can also implement the above approach:-

maps gives you an even fancier way to deal with objects. Very similar to **Maps** in C++

```
interface User {
  id: string;
  name: string;
}

// Initialize an empty Map
const usersMap = new Map<string, User>();

// Add users to the map using .set
usersMap.set('abc123', { id: 'abc123', name: 'John Doe' });
usersMap.set('xyz789', { id: 'xyz789', name: 'Jane Doe' });

// Accessing a value using .get
console.log(usersMap.get('abc123')); // Output: { id: 'abc123', name: 'John Doe' }
```

## **Exclude :-**

In a function that can accept several types of inputs but you want to exclude specific types from being passed to it.

```
type Event = 'click' | 'scroll' | 'mousemove';
type ExcludeEvent = Exclude<Event, 'scroll'>; // 'click' | 'mousemove'

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

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

When using zod, we're done runtime validation.

For example, the following code makes sure that the user is sending the right inputs to update their profile information

```
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(),
});

app.put("/user", (req, res) => {
  const { success } = userProfileSchema.safeParse(req.body);
  const updateBody = req.body; // how to assign a type to updateBody?

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

app.listen(3000);
```

