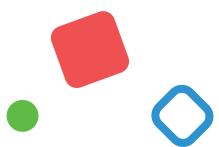




TypeScript-Baby

by Lwin Moe Paing





မိတ်ဆက်

ဒီစာအပ်က TypeScript ကိုအခုံမှစလေ့လာမယ့်သူတွေအတွက် အခြေခံကြမယ့်
အကြောင်းအရာတွေပဲဖြစ်ပါတယ်။ ဒီစာအပ်က လုံးဝ Beginner တွေအတွက်
ရည်ညွှန်းထားတာဖြစ်တဲ့အတွက်၊ အခုံလက်ရှိ Beginner Level ကနေ နောက်ပိုင်း
ကိုယ့်ဘာသာ ဆက်လက်လေ့လာသွားနိုင်ဖို့အတွက် ရည်ရွယ်ထားတာဖြစ်ပါတယ်။
စတင်လေ့လာမယ့်သူများအတွက် JavaScript, Npm အကြောင်းအရာတွေကို သိရှိပြီး
သားဖြစ်နေမှ ဒီစာအပ်ကို လေ့လာသွားလို့ရမှာဖြစ်ပါတယ်။ Typescript Beginner များ
စွာအတွက် များစွာအထောက်အကျပြုစေလိမ့်မယ်လို့ လည်း ယုံကြည်ပါတယ်။

စာရေးသူအကြောင်း

ကျွန်တော်နာမည်ကိုလွှင်မိုးပိုင် လို့ခေါ်ပါတယ်။ ယခု စာရေးနေစဉ်ကာလမှာ ကျွန်တော်
က Web Development ပတ်ဝန်းကျင်မှာ ကျင်လည်ခဲ့တာ (၆) နှစ်ကျော် ကာလသို့
ရောက်ရှိခြုံပြီဖြစ်ပါတယ်။ ယခုလက်ရှိ Software House ကုမ္ပဏီတစ်ခုမှာ Senior
Frontend Developer အနေနဲ့ လုပ်ကိုင်လျက်ရှိပြီး။ အပြင် Project ပေါင်းများစွာကို
လည်း လက်ခံရေးသားပေးလျက်ရှိပါတယ်။ ယခင်က ကျွန်တော် Binary Lab (2023,
2024)၊ uab bank (2022) နဲ့ MoMoney (powered by ShweBank) တို့မှာ React
Native Mobile Developer အဖြစ်လည်း တာဝန်ယူလုပ်ဆောင်ခဲ့ဖူးပါတယ်။ Mar-
athon Myanmar Devliery Co, မှာလည်း Senior Vue Developer အနေနဲ့ 2020
တုန်းက လုပ်ကိုင်ဆောင်ချက်ခဲ့ပါတယ်။ UCSY ကို 2013 မှာ တက်ခဲ့ပြီး တစ္ဆေးသို့လဲ
ကျောင်းသားသာဝတုန်းက ရရှိခဲ့တဲ့ Achievements တွေကတော့ Unihack Hackathon
(2018) မှာ Winner နဲ့ Japan-Myanmar ပူးပေါင်းစီစဉ်သော Wit Design Award
(2018) ကိုရရှိခဲ့တာဖြစ်ပြီး၊ နောက်ထပ် Wit Champion Award ကို (2020) မှာ ရရှိခဲ့ပါ
တယ်။ အခုံ စာအပ်ကတော့ ကျွန်တော့ရဲ့ခုတိယမြောက် Mini Ebook စာအုပ်လေးဖြစ်
ပါတယ်။

စာရေးသူအား ဆက်သွယ်လိုပါတာ - lwinmoepaing.dev@gmail.com (သို့)
ဖုန်းနံပါတ် - 09420059241 ကို ဆက်သွယ်နိုင်ပါတယ်။





လိုင်စင်

အခကြေးငွေဖြင့် ရောင်းခြခြင်း ပြုလုပ်လို့မရပါ။

Credit ပေးပြီး Share လုပ်နိုင်ပါသည်။ Non Commercial အမျိုးအစားဖြစ်သည်။
Original Same License အောက်မှ Contribution များလုပ်ဆောင်နိုင်ပါသည်။

<https://creativecommons.org/licenses/by-nc-sa/4.0>

TypeScript Baby by Lwin Moe Paing @ 2025

licensed under CC BY-NC-SA 4.0





Table of Contents

TypeScript ဆိတ်ဘာဘာလဲ ?	1
Preparing for Coding	3
NodeJS Installation	3
TypeScript Package Installation	4
Install typescript package	4
Install ts-node package	5
TypeScript Config	5
Manual Config	5
Config Init with tsc	9
Basic TypeScript	9
JavaScript Type ချုပ်	9
JS String	9
JS Number	9
JS Boolean	9
JS Undefined	10
JS Function	10
JS BigInt	10
JS Symbol	10
JS Object	10
TypeScript Annotation vs Inference	11
TypeScript Basic Syntax	11
TypeScript Annotation	11
TypeScript Inference	12
Inference နဲ့ Annotation ကို ဘယ်အချင့်မှာ ဘယ်ဟာသုံးရမှာလဲ	13
TS - String	13
TS - Number	14
TS - Boolean	14
TS - Null	14





TS - Undefined	15
TS - Union & Literal	15
Type Level တွေကို ခွဲခြားမယ်	15
Union (OR)	16
Literal Union	17
Literal Union နဲ့ မှားတတ်သောအရာ	18
TS - Unknown & Any	20
Unknown & Any	20
Any Type	21
Unknown Type	21
TS - Never	22
TS - Enum	23
TS - Array	26
TS - Object	27
Object Type Annotation	28
Optional Properties	29
TS - Type keyword	30
TS - Tuple	32
Turple	32
React useState hook tuple	34
TS - Function	34
Function Type	34
Arrow Function (=>)	35
Normal Function	36
Custom Type Function	37
Function Void Type	37
Function Overloading	38
Signature သတ်မှတ်ခြင်း	39
Implementation ပြုလုပ်ခြင်း	39
TS - Intersection	41
Intersection	41





TS - Interface	42
Interface keyword	42
Extending Interfaces	42
Interface As Function	43
TS - readonly	43
Readonly property	44
Readonly Array	44
TS - keyof	45
TS - typeof	46
typeof	46
keyof typeof တွဲပေးခြင်း	49
TS - Index Type	50
Index Signature	50
TS - Predefined & Index	51
TS - Union keys in Index	52
TS - Generic	53
Generic ဘို့ဘာ ?	53
Generic Function	53
Exercise for Generic	56
Generic Type	58
TS - as Type Casting	60
TS - satisfies	61
Utility	65
TS - Pick	65
TS - Omit	66
TS - Partial	67
TS - Required	68
TS - Record	69
TS - Readonly	70
Other Topic	71



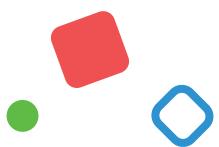


TypeScript Narrowing	71
TypeScript Predicate Guard	73
TypeScript Template Literal	74
နိဂုံး	76





ကျွန်ုင်တော်တို့ TypeScript Baby ကို
စတင်ပါတော့မယ် ...

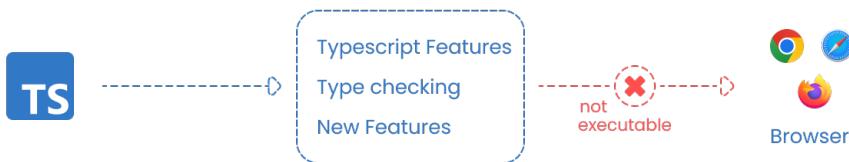




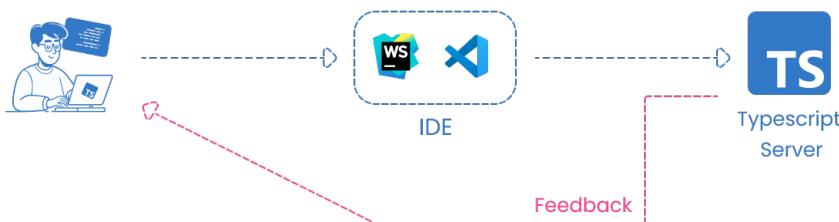
TypeScript ဆိုတာဘာလဲ ?

Programming လောကမှာ JavaScript ဆိုတာ အများဆုံးသုံးတဲ့ programming language တစ်ခုဖြစ်လာပါတယ်။ သို့ပေမယ့်လည်း JavaScript ဟာ Dynamic Type Language (loosely typed) တစ်ခုဖြစ်သည့်အတွက် Development လုပ်နေချိန်အတွင်းမှာ ဖြစ်လာနိုင်တဲ့ error တွေကို အလွယ်တကူ ရှာမထွေ့နိုင်ပါဘူး။

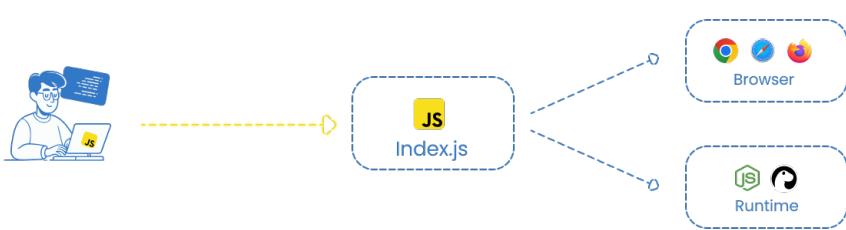
ဒီအတွက် Microsoft ကနေ Typescript ဆိုပြီးတော့ JavaScript ကို အကြခံပြီး error တွေကို ပိုမိုလွယ်ကူစာ ရှာဖွေနိုင်ဖို့ အထောက်အကူပေးတဲ့ superset တစ်ခုကို ဖန်တီးလိုက်ပါတယ်။



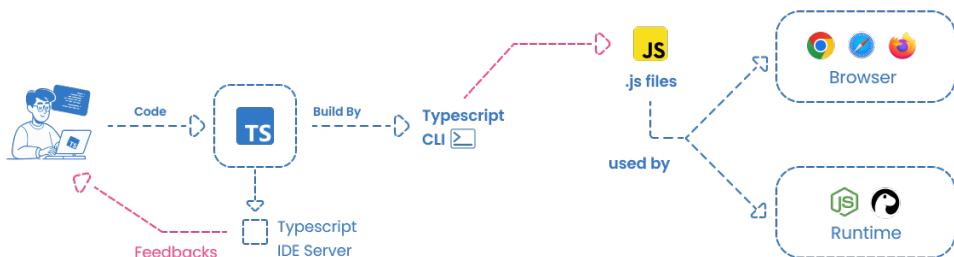
Typescript မှာ Type-checking တွေကို support လုပ်ပေးနိုင်တဲ့အတွက် Data type တွေကို ကျွန်ုတ်တို့ Development လုပ်နေစဉ်မှာပဲ အလွယ်တကူ သိရှိနိုင်ပါတယ်။



TypeScript ကို Support လုပ်တဲ့ IDE တွေရဲ့ နောက်ကွယ်မှာ TypeScript Server ကို Run ထားရင်း Development လုပ်နေတဲ့အချိန်မှာပဲ Feedback တွေကို အချိန်နဲ့ တပြေားညီ ကျွန်ုတ်တို့က သိရှိနိုင်ပါတယ်။



သာမန်အရ ကျွန်တော်တို့က (.js) extension JavaScript ဖိုင်တွေကို Browser တွေ နဲ့ NodeJS, Deno တို့လို JavaScript Runtime Environment တွေပေါ် တင်Run နိုင်ပါ တယ်။



ပုံထဲကအတိုင်း TypeScript ဖိုင်ကနေ JavaScript ဖိုင်ကိုပြန်ပြောင်းရပါတယ်။ ကျွန်တော်တို့ TypeScript CLI နဲ့ Compile လိုက်တဲ့ အခါ JavaScript ဖိုင်အဖြစ် ပြောင်းလဲသွားပြီးမှ Browser တွေနဲ့ Runtime Environment တွေမှာ အသုံးပြုလိုရမှာ ပါ။ ဒီစာအုပ်ထဲမှာ TypeScript ကို အသုံးပြုမယ့် အကြောင်းအရာတွေကို လေ့လာ သွားကြရအောင် ...။



Preparing for Coding

NodeJS Installation

ပထမ္မားဆုံး မိမိတို့စက်ထဲမှာ NodeJS ကို ထည့်သွင်းထားဖို့လိုအပ်ပါတယ်။
မရှိသေးပါက <https://nodejs.org/> မှာ သွားပြီး NodeJS ကို Download ဆွဲပြီး install လုပ်ပေးပါ။

Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

Download Node.js (LTS)

Downloads Node.js v22.13.1 with long-term support. Node.js can also be installed via package managers.

Want new features sooner? Get [Node.js v23.6.1](#) instead.

1 // server.mjs
2 import { createServer } from 'node:http';
3
4 const server = createServer((req, res) => {
5 res.writeHead(200, { 'Content-Type': 'text/plain' });
6 res.end('Hello World!\n');
7 };
8
9 // starts a simple http server locally on port 3000
10 server.listen(3000, '127.0.0.1', () => {
11 console.log('Listening on 127.0.0.1:3000');
12 });
13
14 // run with 'node server.mjs'

JavaScript Copy to clipboard

Learn more what Node.js is able to offer with our [Learning materials](#).

ကိုယ့်စက်ထဲ သွင်းပြီးပါက Terminal ထဲမှာ npm -v ရိုက်ပြီး Node Package Manager ရဲ့ Version ကို စစ်ဆေးနိုင်ပါတယ်။

```
npm -v
```

version နံပါတ်ပေါ်လာပါက သွင်းပြီးကြောင်းသိနိုင်ပါတယ်။



TypeScript Package Installation

Install typescript package

TypeScript ဖိုင် တွေကို Browser , Runtime တွေကနေ တိုက်ရှိက် အလုပ်မလုပ်နိုင် တဲ့အတွက် JavaScript ဖိုင်ကိုပြန်ပြောင်းဖို့ လိုပါတယ် ။ ဒီလို ပြောင်းဖို့အတွက် ပထမ ဦးစွာ typescript package ကို globally install ပြုလုပ်ပါမယ် ။

```
npm i typescript -g
```

typescript package ကို install လုပ်ပြီးသွားတဲ့အခါ tsc ဆိတ္တဲ့ command ကို အသံးပြု လိုရသွားမှာပါ ။ tsc command က .ts ဖိုင်တွေကို .js ဖိုင်အဖြစ် ပြောင်းလဲပေးနိုင် ပါတယ် ။ ဥပမာ အနေနဲ့ index.ts ဆိတ္တဲ့ typescript ဖိုင်တစ်ခု ကိုအရင် ပြုလုပ်ပြီး အောက်ပါ Code ကို Save လိုက်ပါမယ် ။

```
const age: number = 20;
console.log(age);
```

ဒီနောက် Terminal ကိုဖွင့်ပြီး tsc command ကိုရိုက်ပြီး js ဖိုင်ကို ထုတ်ကြည့်ပါ ။

```
tsc index.ts
```

ဒါဆိုရင် ကိုယ့်စက်ထဲမှာ index.js ဆိုပြီး ဖိုင်အသစ်တစ်ခု အောက်က Code အတိုင်း ထွက်လာတာကို တွေ့ရမှာပါ ။ ထူးစွားချက်အနေနဲ့ typescript မှာသုံးခဲ့တဲ့ number type ကိုဖြတ်ပြီး const နေရာမှာ var ကို ပြောင်းလဲထုတ်ပေးလိုက်တာကို တွေ့ရမှာပါ ။

```
var age = 20;
console.log(age);
```

အခုလိုပုံစံနဲ့ tsc command က typescript code တွေကို js code တွေအဖြစ် compile လုပ်ပြီး ပြောင်းလဲပေးပါတယ် ။



Install ts-node package

ဒုတိယ package အနေနဲ့ ts-node ကိုလည်း globally install ပြုလုပ်ပါမယ်။

```
npm i ts-node -g
```

package ကို install လုပ်ပြီးသွားတဲ့အခါ ts-node ဆိုတဲ့ command ကို အသုံးပြုလို့ရ သွားမှာပါ။ ဒီ command က .ts ဖိုင်တွေကို runtime မှာ တိုက်ရှိက် execute လုပ်ပေး နိုင်ပါတယ်။ .js ဖိုင် မကြောင်းတော့ဘဲ အချိန်ကုန်သက်သာပြီး code debugging လုပ်တဲ့ အခါ သုံးကြပါတယ်။

TypeScript Config

Manual Config

TypeScript စလုပ်တော့မယ် ဆိုရင် ချုန်ထားလို့မရတာက config ချေတာပဲဖြစ်ပါတယ်။

001-ts-compile-to-js နာမည်နဲ့ ဖိုလ်ဒါ အသစ်တစ်ခု အသစ်တစ်ခု အတွင်းမှာ index.ts ရယ်၊ math.js ရယ်၊ tsconfig.json ဆိုတဲ့ ဖိုင် (၃) ဖိုင် ပြုလုပ်ပါမယ်။

tsconfig.json ထဲမှာ အောက်က code ကို save လိုက်ပါမယ်။

```
{
  "compilerOptions": {
    "strict": true,
    "target": "ES5",
    "module": "ESNext",
    "outDir": "./dist"
  }
}
```

math.ts ဖိုင်ထဲမှာ add function code ကို ရေးသားလိုက်ပါမယ်။



```
const add = (a: number, b: number) => {
    return a + b;
};

export default add;
```

index.ts ဖိုင် ထဲမှာ အောက်တာ code ကို save လိုက်ပါမယ်။

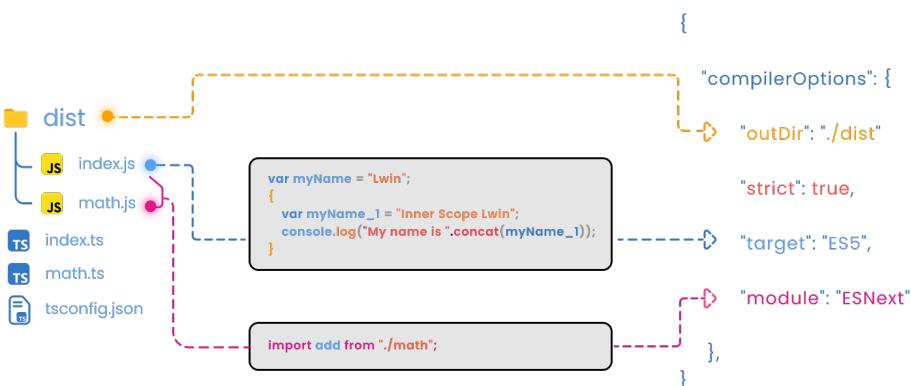
```
import add from "./math";

const myName = "Lwin";

{
    const myName = "Inner Scope Lwin";
    const output = `My name is ${myName}`;
    console.log(output);
}

add(1, 2);
```

စာဖတ်သူက tsc ဆိုတဲ့ command နဲ့ Terminal မှာ စမ်းလိုက်လိုရပါတယ်။ ဒါနိုင် dist ဆိုတဲ့ ဖိုလ်ဒါထဲမှာ javascript ဖိုင်တွေအဖြစ် ရောက်သွားတာကိုတွေ့ရမှာပါ။





tsconfig.json ဆိုတဲ့ ဖိုင်က config ချတဲ့ ဖိုင်ဖြစ်ပါတယ်။

ဒီအပိုင်းမှာ compilerOptions ကို အဓိကထား လေ့လာသွားပါမယ်။

ပထမဆုံးတွေ့ရတဲ့ outDir ဆိုတဲ့ Property အဓိပ္ပာဇာည် .js ဖိုင်တွေကို ဘယ်ဖိုလ်ဒါ (Directory) ထဲမှာ ပြုလုပ်မယ်(ထွက်ရှိမယ်) ဆိုတာ သတ်မှတ်ခြင်း ဖြစ်ပါတယ်။

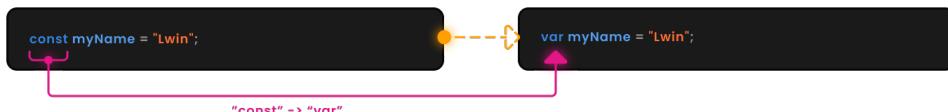
strict ကို true လုပ်ထားတာ (Strict Type-Checking) ကို ဖွင့်ထားမယ်လို့ ဆိုလိုတာပါ။ undefined , null တို့ကို သုံးတဲ့နေရာ မှန်ကန်ဖို့ရယ်။ Code Structure ကို ကျင့်ကျစ် လစ်လစ် ရှုံးချင်တဲ့ အတွက် အသုံးပြုကြ ပါတယ်။

target ထဲမှာ ES5 ကို ရွေးချယ်ထားပါတယ်။ အဓိပ္ပာဇာ ထွက်လာမယ့် .js ဖိုင် တွေသည် Escma 5 version အဖြစ်သို့ ပြုလုပ်မယ်လို့ သတ်မှတ်ထားတာပါ။

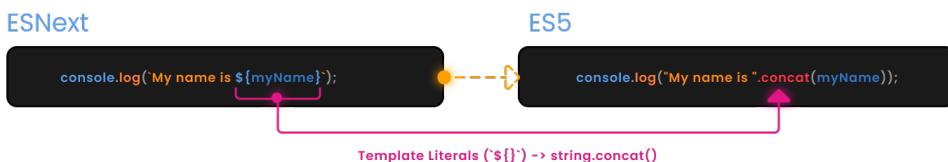
2009 ခုနှစ်မှာ ECMAScript 5 (ES5) က Release လုပ်ဖြစ်ခဲ့ပါတယ်။ အခု Modern JavaScript ရေးတုံးတွေမှာ တွေ့ရတဲ့ let const တို့လို့ scope variable တွေ မရှိသေးပါဘူး။ var တစ်လုံးပဲ သုံးလို့ရပါသေးတယ်။

2015 မှာ ထွက်ရှိလာတဲ့ ECMAScript 6 ကျမှပိုပြီး Modern နဲ့ Readable ဖြစ်စေမယ့် Feature တွေ syntax တွေနဲ့အတူ ထွက်ရှိလာပါတယ်။ ဥပမာ အားဖြင့် Arrow function =>, Template Literals, Destructuring လုပ်တာတွေပါ လာပါတယ်။

ESNext



ES5



ES5

မှတ်ချက်။ ၂၀၁၅ မှာထွက်တဲ့ ES6 ကို ES2015 လို့လည်းခေါ်ခေါ်ကြပါတယ်။ ဒါ ကြောင့် ES6 နဲ့ ES2015 က အတူတူပဲ ဖြစ်ပါတယ်။



```
{
  "compilerOptions": {
    // ... other configs
    "target": "ESNext",
  }
}
```

စာဖတ်သူက target ရဲ့ တန်ဖိုးကို ESNext လို့ပြောင်းပြီး tsc command နဲ့ JavaScript ဖိုင်ကို ပြောင်းကြည့်ပါ။ ဒါဆိုရင် ထွက်လာမယ့် ဖိုင်တွေထဲမှာ var တွေမဟုတ်တော့ဘဲ Modern JavaScript ရေးထုံးတွေကို Support သွားမှာဖြစ်ပါတယ်။

target ရဲ့ တန်ဖိုးနေရာမှာ “ES5”, “ES6”, “ES2015”, ကနေ လက်ရှိ “ES2024” အထိ ကိုယ်ပြောင်းလဲချင်တဲ့ Version ရွေးချယ်လို့ရပါတယ်။ ESNext ဆိုတာကတော့ နောက်ဆုံးလက်ရှိ ရောက်နေတဲ့ Version ကို ဆိုလိုချင်တာပါ။

module ထဲမှာ ESNext ကို ရွေးချယ်ထားပါတယ်။ ES6 ကနေစပြီးတော့ import နဲ့ export တို့ကို စပြီး Support နေပါပြီ။ ကျွန်ုတ်တို့က Node.js Environment အတွက်ပဲ ရေးသားမယ်ဆိုရင် CommonJS ကိုပြောင်းပြီး သုံးနိုင်ပါတယ်။

ESNext Module

```
import add from "./math";
```

CommonJS Module

```
var math = require("./math");
```

import -> require

အခြား ရွေးချယ်စရာ Module Type တွေအတွက် “UMD”, “AMD”, “System” တို့ရှိပါ သေးတယ်။ သို့ပေါ်မယ် ဒီစာအုပ်မှာတော့ အဲ Module type တွေတော့ အကျယ်တစ်မျိုး မရေးသွားတော့ပါဘူး။



Config Init with tsc

အရင်ဆုံး တည်ဆောက်ခဲ့တဲ့ tsconfig.json ကို delete လိုက်ပါမယ် ။ tsc command နဲ့ initialize လုပ်ပါမယ် အောက်က code ကို terminal မှာ စမ်းပေးပါ ။

```
tsc --init
```

ဒါနိုင် tsconfig.json ကို အဂိုအလောက် တည်ဆောက်သွားမှာဖြစ်ပြီး၊ အထူက code အတွင်းမှာလေ့လာလို့ရမယ့် <https://aka.ms/tsconfig> လင့်ကို ဖော်ပြထားမှာဖြစ်ပါတယ် ။

Basic TypeScript

JavaScript Type များ

TypeScript ရဲ့ Type တွေကို မသွားခင် ။ typeof ကိုသုံးပြီး JavaScript Type တွေကို အတူတူ ပြန်နေးရအောင် ။

JS String

```
typeof "Hello World !!"
```

string

JS Number

```
typeof 1000
```

number

JS Boolean

```
typeof true
```

boolean



JS Undefined

```
typeof undefined
```



undefined

JS Function

```
typeof function () { }
```



function

JS BigInt

```
typeof BigInt("145")
```



bigint

JS Symbol

```
typeof Symbol("@")
```



symbol

JS Object

```
typeof null
```

```
typeof {}
```

```
typeof ["hello", "world"]
```

```
typeof new Map()
```

```
typeof new Set()
```



object





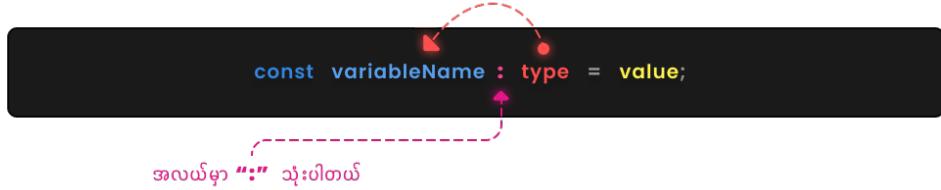
TypeScript Annotation vs Inference

Annotation vs Inference ကို အလွယ်ရှင်းပြရရင်

1. Type Annotation Type ကို manually သတ်မှတ် ခြင်း
2. Type Inference Type ကို TypeScript က auto ယူဆခြင်း

တို့ပဲ ဖြစ်ပါတယ် !!

TypeScript Basic Syntax



အကြခံအရ (:) column ကိုအသုံးပြုပြီး အနောက်မှာ type ကိုသတ်မှတ်ပါတယ် !!

TypeScript Annotation

Type ကို manually သတ်မှတ်ခြင်း code ကို လေ့လာကြည့်ကြပါမယ် !!

```
let myName: string = "LMP";  
  
let age: number = 28;  
  
let isStudent: boolean = true;
```

Programmer က Type ကို ဂိုလ်ဂိုင်စိမ်ရေးသားတာ ဖြစ်ပါတယ် myName variable ထဲကို string Type ပဲထည့်ပါမယ်လို့ ကြော်လားတာ ဖြစ်ပါတယ် !! age ထဲကို number Type ဖြစ်ပြီး ၅ isStudent ထဲမှာ boolean Type အဖြစ်နဲ့ သုံးထားပါတယ် !!



Type Annotation က variable, function parameter, return type, class property တွေမှ သတ်မှတ်နိုင်ပါတယ်။ နောက်ပိုင်းအခန်းတွေမှာ ဆက်ပြီး လေ့လာ အသုံးပြုသွားမှာပါ။

TypeScript Inference

TypeScript က သင့်တော်တဲ့ Type ကို အလိုအကျောက် အသုံးပြုပေးတာကို Inference လို့ခေါ်ပါတယ်။

```
let age = 25;

let name = "John";
```

Type ကို ကိုယ်တိုင် (Manual) သတ်မှတ် ပေးစရာမလိုပါဘူး။ age variable ထဲကို တန်ဖိုး 25 ထည့်လိုက်တဲ့အခါ TypeScript က အလိုအလောက် number type ဖြစ်သတ်လို့ သတ်မှတ်ပေးလိုက်ပါတယ်။

The screenshot shows a code editor window for 'index.ts'. The code contains three lines:

```
1 let age = 25
2
3 console.log(age);
```

A tooltip is displayed over the second 'age' declaration, which reads:

age variable မီ Mouse နဲ့
အကြောက်လေး တင်ထားလိုက်ရင်
number type ဖြစ်ကြောင်းကို တွေ့ရမှာပါ။

თာဖတ်သူက age variable ကို Mouse နဲ့ ခက္ကကြာကြားလေး ပြမ်ထားရင် သူ့ရဲ့ Type ကို မြင်နိုင်ပါတယ်။

နောက်တစ်လိုင်းမှာ age = 'string'; string တန်ဖိုးထည့်လိုက်ရင် TypeScript server က အနေ Error Feedback ပြန်ပေးပါလိမ့်မယ်။



```
index.ts 1, U •
index.ts > ...
1 let age = 25;
2
3 console.log(age);
4
5 Type 'string' is not assignable to type 'number'. ts(2322)
6 △ Error (TS2322) ⚠ | ⚡
7 Type string is not assignable to type number
8
9 let age: number
10
11 View Problem (CF8) No quick fixes available
12 age = "string";
13
14
```

age variable ကို number type ဖြစ်နေပြီမိန့်လို့
string စနစ်းကို ထည့်လို့မရတဲ့အတွက်
Error ဖော်တော်ဖြစ်ပါတယ်။

ဒါဆို TypeScript ဘယ်လိုအလုပ်လုပ်သလဲ Inference နဲ့ Annotation ဆိုတာဘာလဲ
အကြခံအားဖြင့် သိသွားပါပြီ။

Inference နဲ့ Annotation ကို ဘယ်အချင်မှာ ဘပ်ဟာသုံးရမှုလဲ

ပုံမှန်အားဖြင့် Primitive Data Type တွေကိုသုံးတဲ့အခါ Inference (အလိုအလောက်
သတ်မှတ်ပေးခင်း) ကိုသုံးလေ့ရှုပါတယ်။

Primitive Types (7 မျိုး) - string, number, boolean, bigint, symbol, null, undefined
တို့ဖြစ်ပါတယ်။

အခြား Complex ဖြစ်တဲ့ Object Type တွေကိုတော့ Annotation သုံးလေ့ရှုတြဲပါတယ်

TS - String

```
const myName : string = "Lwin Moe Paing";
```

let myName : string = "str value" ရေးထုံးနဲ့ String Type တွေကို သတ်မှတ် လို့ရပါ
တယ်။



TS - Number

```
const myId : number = 1;
```

let year: number = 2024; ရေးထံးနဲ့ Number Type တွေကို သတ်မှတ် လိုပါတယ်။

TS - Boolean

```
const isPerson : boolean = true;
```

let isPerson : boolean = true; ရေးထံးနဲ့ Boolean Type တွေကို သတ်မှတ် လိုပါတယ်။

TS - Null

```
const nullData : null = null;
```

let nullData : null = null; ရေးထံးနဲ့ Null Type တွေကို သတ်မှတ် လိုပါတယ်။
တကယ်တော့ ဒါဟာ အဓိပ္ပာယ်မရှိပါဘူး null type ကိုကြော်ပြီး ဘယ်သူမှ လက်တွေ၊
မှာ ရေးသားလေ့မရှိဘူး။ ဖြစ်လေ့ ဖြစ်ထရှိတဲ့ အမှားက null type inference ပဲဖြစ်ပါတယ်။

```
let nullInfer = null;
```

any type (Type inference)

let nullInfer = null; ကိုယ်တိုင် မသတ်မှတ်ပဲ inference ပုံစံ null value ကို ထည့်လိုက်တဲ့အခါ TypeScript က any type အဖြစ် အလိုအလောက် သတ်မှတ်သွားမှာပါ။



TS - Undefined

```
const undefinedData : undefined = undefined;
```

let undefinedData : undefined = undefined; ရေးထုံးနဲ့ undefined Type တွေကို
သတ်မှတ် လို့ရပါတယ်။ တကယ်တော့ ဒါဟာလည်း အဓိပ္ပာယ်မရှိပါဘူး။

```
let undefinedInfer = undefined;
```

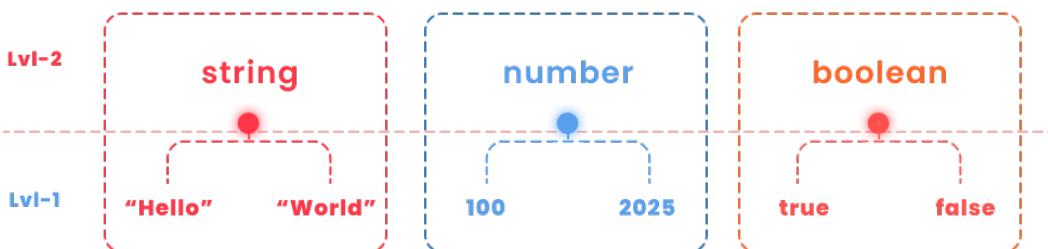
any type (Type inference)

let undefinedInfer = undefined; ကိုယ်တိုင် မသတ်မှတ်ပဲ inference ပုံစံက any type
အဖြစ် အလိုအလျောက် သတ်မှတ်သွားမှာပါ။

TS - Union & Literal

Type Level တွေကို ခွဲခြားမယ်

Union အကြောင်းမသွားခင် အောက်က ပုံလေးနဲ့ Level လေးတွေ လေ့လာကြည့်ပါမယ်



ပထမ Level 2 မှာရှိတဲ့ string type ကို သတ်မှတ်ထားမယ်ရင်ဆို စာသား "value" (သို့မဟုတ်) 'value' စာသားမှန်သမျှကို လက်ခံနိုင်ပါပြီ။ အတူ number type အတွက် နံပါတ် အားလုံးကို လက်ခံနိုင်ပါတယ်။ boolean type အတွက် true (or) false တန်ဖိုးများကို ထည့်သွင်းနိုင်ပါတယ်။

Union | (OR)

Type တစ်ခုထက် ပိုပြီး လက်ခံချင်တဲ့အပါ Union | (OR) ကို အသုံးပြုပြီး လက်ခံလို့ရပါတယ်။

```
let stringOrNumber : number | string = "Hello";
```

(၅) syntax ကိုယ်ပါတယ်

နမူနာ အရ let stringOrNumber: number | string ဆိုပြီး ရေးသားထားပါတယ်။ string တန်ဖိုးကော့ number တန်ဖိုးကိုပါ ထည့်သွင်းအသုံးပြုနိုင်ပါတယ်။

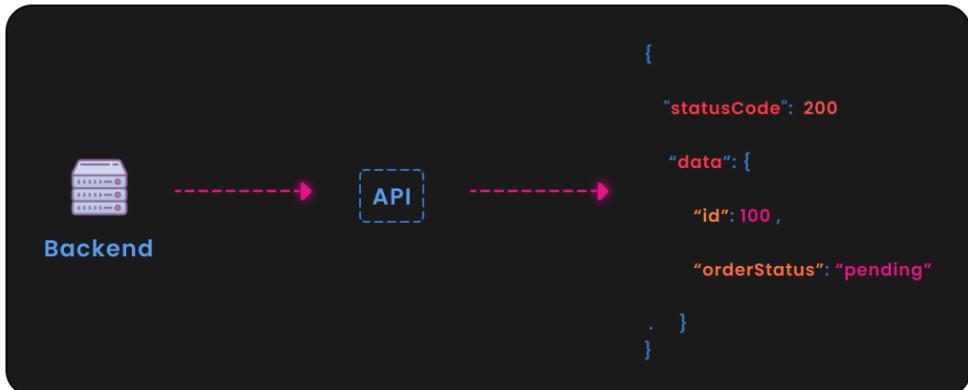
စာဖတ်သူတို့က boolean type ပါ အသုံးပြုချင်တယ်ဆိုရင် ဘယ်လို ရေးရမလဲ အကြောက်တော့ အောက်ပါအတိုင်းဖြစ်သွားမှာပါ။

```
let strOrBoolOrNo: string | number | boolean = true;
```



Literal Union

Union ကို အခြေခံအားဖြင့် ရသွားပါပြီ။ သို့ပေါ်မယ် တကယ့်လက်တွေ့မှာ Literal Union ကို အသုံးများပါတယ်။ Literal ဆိုတာက တိတိ ကျကျ သတ်မှတ်ထားတဲ့ တန်ဖိုး တွေကိုပြောချင်တာပါ။



နမူနာပုံထဲကအတိုင်း Backend Server ကနေ API JSON ရလာပါမယ်။ အထူးမှာ statusCode အနေနဲ့ 200 ရလာပြီး orderStatus အနေနဲ့ "pending" ကို ရလာပါတယ်။



Backend Documentation အရ orderStatus တန်ဖိုးသည် pending , delivered , success ပဲရမယ်။ statusCode တန်ဖိုးသည် 200 သို့မဟုတ် error တစ်ခုခုဖြစ်ခဲ့ရင် 400 ပြန်မယ်လို့ ရေးထားမယ်။

Type အရ orderStatus type ကို string ပဲ လက်ခံတာထက် တိတိကျကျ တန်ဖိုး pending , delivered , success ကိုပဲ လက်ခံတာကောင်းပါတယ်။ ဒါမျိုး TypeScript မှာ Literal Union ပုံစံနဲ့ ရေးလို့ရပါတယ်။



```
let orderStatus : "pending" | "delivered" | "success" = "pending";
```

တိကျသော တန်ဖိုးများရှိ Literal Type လိုပေါ်တယ်

let orderStatus: "pending" | "delivered" | "success" ပုံစံနဲ့ Literal Union Type ရေးသားလိုက်တာပါ။

The screenshot shows a code editor window for 'index.ts'. The code is:

```
1 let orderStatus: "pending" | "delivered" | "success" = "pending"
2
3 orderStatus = "delivered"
```

A cursor is at the end of 'delivered'. A dropdown menu shows three options: 'delivered', 'pending', and 'success'. A callout bubble points to the 'delivered' option with the text 'Suggestion ပေးအနေဖြစ်ပါတယ်။' (Suggestion given as an example).

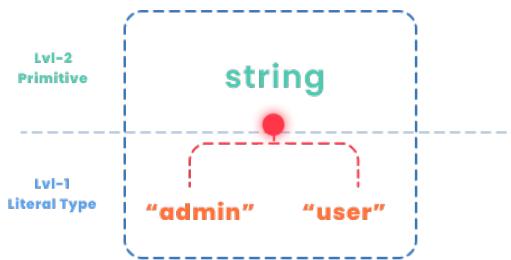
တာဖတ်သူတို့က statusCode အတွက် တန်ဖိုး 200 နဲ့ 400 ပဲလက်လို့ရမယ့် type variable တစ်ခု တည်ဆောက်ကြည့်ပေးပါ။

အဖြောက်တော့ အောက်ပါ Code အတိုင်း ရလာမှာဖြစ်ပါတယ်။

```
let statusCode: 200 | 400 = 200 ;
```

Literal Union နဲ့ မှားတတ်သောအရာ

ပထမဆုံး Level ခွဲထားတဲ့ ပုံကို မှတ်မိုးမယ်ထင်ပါတယ်။ Level 2 Primitive Type - String က အောက်က Literal Type ထက် Priority ပိုမြင့်ပါတယ်။



Literal union type + Primitive type ကို အတူတူရေးသားလိုက်မယ်ဆိုရင် priority မြင့် စွဲ Type က Overwrite သွားမှပါ။



String က Priority မြင့်တဲ့အတွက် စာသားတန်ဖိုး အားလုံးကို လက်ခံသွားနိုင်ပါတယ်။ admin နဲ့ user Literal တွေကို Suggestion လည်းပေးမှာမဟုတ်တော့ပါဘူး။

```
index.ts  X  tsconfig.json
index.ts > ...
1  let role: "admin" | "user" | string = "admin";
2  ^
3  role = "";  ←-----  String က ပိုမို: Priority မြင့်တဲ့အတွက် စာသားအားလုံးကို လက်ခံနိုင်ပြီ: Suggestion မပေးတော့ပါ။
4
```

ဒီလောက်ဆို Union အကြောင်း နားလည်ပြီလို့ လို့ယူဆပါတယ်။



TS - Unknown & Any

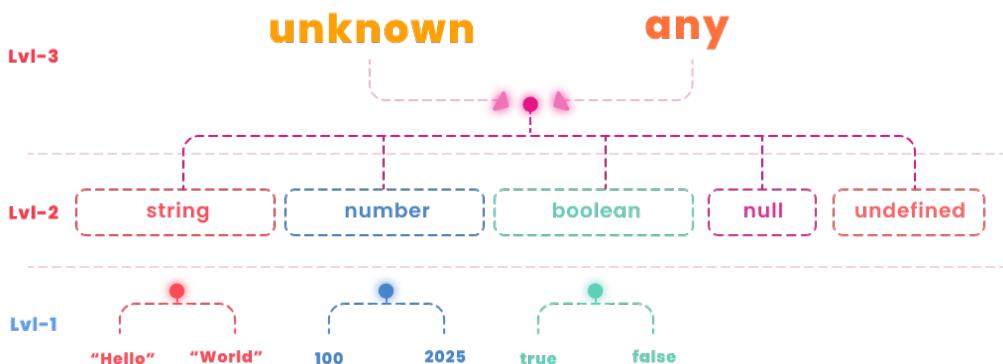
Unknown & Any

```
const variableName : unknown = value;
const variableName : any = value;
```

let unData : unknown = "hello"; ရေးထံးနဲ့ unknown Type တွေကို သတ်မှတ် လိုပါပဲ တယ်။

let anyData : any = "hello"; ရေးထံးနဲ့ any Type တွေကို သတ်မှတ် လိုပါတယ်။

Priority Level ကို အရင် ကြည့်ရအောင်။



unknown နဲ့ any type ၏ Priority Level အမြင့်ဆုံးတွေဖြစ်ပါတယ်။

ဒီ Type တွေကို သတ်မှတ်ထားတဲ့အခါ အေားတန်ဖိုးတွေအကျိုး လက်ခံနိုင်ပါတယ်။



Any Type

ဘယ်လို Data type မဆို ကိုင်တွယ်နိုင်တဲ့ Flexible Type ဖြစ်ပါတယ်။

Strict Type-Checking ကို Disable လုပ်နိုင်စွမ်းရှိပါတယ်။

```
index.ts
1 let anyAnnotation: any = 123;
2 anyAnnotation = true;
3
4 anyAnnotation = "Something";
5
6 any
7 anyAnnotation.nothingMethod();
```

Any Type ဆဲတို့ ဘယ်အပျိုးအစားမလိုလက်ခဲ့လို့ရပါတယ်။

မရှိတဲ့ Method (သို့) Properties ထောက်လေးလေးတွေကို TypeScript Error Feedback မပေးတော့ပါ။

any Type ကို တတ်နိုင်သလောက် ရောင်းပြီး သုံးသင့်ပါတယ်။

သို့ပေါ့မယ် လည်း Legacy JavaScript Code တွေကို Migrate လုပ်တဲ့အခါ။

API response ကို Handle လုပ်တဲ့အခါမှာ any Type ကို အသုံးပြုလေ့ရှိကြပါတယ်။

Unknown Type

TypeScript မှာ unknown ကို any အစား Type-Safe အနေနဲ့ သုံးကြပါတယ်။

သူလည်းပဲ အခြား Type အားလုံးကို Assign လုပ်လို့ရပါတယ်။ သို့ပေါ့မယ် any နဲ့ မတူတောက method properties တွေကို တိုက်ရှိက် ယူသုံးလို့ မရတာပဲဖြစ်ပါတယ်။

```
index.ts
1 let unknownData: unknown;
2
3 unknownData = true;      // Boolean assign လို၍
4 unknownData = 42;        // Number assign လို၍
5 unknownData = "Hello";   // String assign လို၍
6
7
8 unknownData.length      // Method ဆွဲတဲ့ Direct ခေါ်သုံးလို့မရ။
```

Method (သို့) Properties ထောက်ရှိက် ယူသုံးလို့မရပါ။



TS - Never

```
let neverData : never ;
```

let neverData : never; ရေးသားနည်းနဲ့ never Type တွေကို သတ်မှတ် လိုပါတယ်။

ထူးခြားတာက Never Type ထဲကို ဘယ်တန်ဖိုး အမျိုးအစားမှ Assign လုပ်လို့တွေ့မရပါဘူး။

index.ts 1, U

```
1 let neverData: never;
2
3 neverData = "Hello";
4
5 Type '"Hello"' is not assignable to type 'never'. ts(2322)
6 △ Error (TS2322) [ ] [ ]
7 Type "Hello" is not assignable to type never .
```

Never Type ထဲကို ဘယ်တန်ဖိုး အမျိုးအစားမှ Assign လုပ်လို့မရပါ။

ဒါနို့ Assign လို့လည်းမရဘဲ ဘာလို့ never type ကို လေ့လာနေရတာတော်းလို့ ငြော စရာရှိလာပြီ။ ကျွန်ုတ်တို့ကိုယ်တိုင် Assign မလုပ်ပေမယ့်လည်း အခြား အခြေအနေ တွေမှာ never type တွေကို တွေ့လာနိုင်ပါတယ်။ ဥပမာ Error Throw လိုက်တဲ့ Function ရဲ့ Return Value Type သဲ never ဖြစ်ပါတယ်။

index.ts U

```
1 const throwError = (message: string) => {
2   throw new Error(message);
3 };
4
5 throwData is declared but its value is never read. ts(6133)
6 △ Error (TS6133) [ ] [ ]
7 throwData is declared but its value is never read.
8
9 const throwData: never
10
11 Quick Fix... (⌘.)
12
13 const throwData = throwError("Make Error");
```

Never Type အနေနဲ့
Return ပြန်မှာပါ။

ဘယ်တော့ မ မပြီးဆုံးနိုင်တဲ့ while loop ကို true conditional ထည့်လိုက်တဲ့ function မျိုးမှာလည်း never type ကိုရရှိလာနိုင်ပါတယ်။

```

index.ts > ...
1 const infiniteLoop = () => {
2   while (true) {
3     // Running Forever
4   }
5 };
6
7
8 const infiniteLoop: () => never
9 const foreverData = infiniteLoop();
10

```

နောက်ဆုံးတစ်ခုကတော့ Switch နဲ့ စစ်တဲ့အခါ default scope ရောက်တဲ့အခါ စစ်လိုက်တဲ့ Data က never type အဖြစ် Type Cast ဖြစ်သွားတာမျိုးကို တွေ့ရမှာပါ။

TS - Enum

```

enum Role {
  USER,           const UserRole = Role.USER;    // => τနိုး: 0
  ADMIN          const adminRole = Role.ADMIN;  // => τနိုး: 1
}

```

Enum (Enumeration) ဆိုတာ မကြောင်းလဲနိုင်တဲ့ Constant Data အစ္စတွေအတွက် လွယ်အောင်လုပ်ပေးထားတဲ့ Type တစ်ခုပါ။ ပြထားတဲ့ ပုံစံကအတိုင်း အရင်ဆုံး enum keyword နဲ့ Enum တစ်ခုကို ပြုလုပ်နိုင်ပါတယ်။ Role မှာ USER ရယ် ADMIN ရယ်ပဲ ရှိတဲ့အတွက် Role Enum တစ်ခုကို လုပ်လိုက်တာပါ။ အဲဒါ Enum တန်ဖိုးကို ယူချင်တဲ့အခါ variable တစ်ခုထဲကို Role.User ဆိုပြီး ယူလိုက်လို့ရပါတယ်။ အလိုအလျောက် သတ်မှတ်ပေးထားတဲ့ တန်ဖိုး 0 ကိုရသွားမှာပါ။



თაဖတ်သူတို့က `Role` ထဲကို နောက်ထက် `SUPERVISOR` နဲ့ `SUPERADMIN` role ကို
ထပ်တိုးကြည့်လိုက်ပြီး `console.log` နဲ့ ပိမိစက်ထဲမှာ အဖွော်ရှုပါပဲ။

```

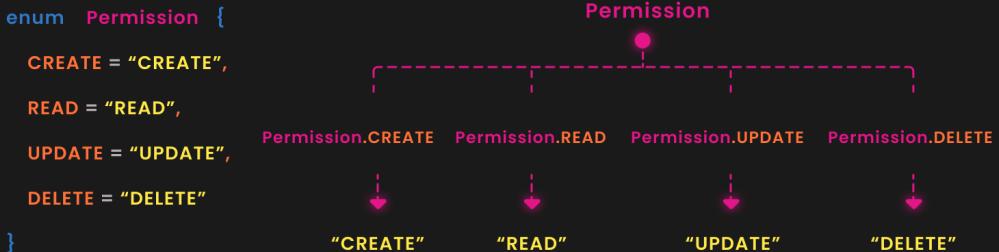
enum Role {
    USER,
    SUPERVISOR,
    ADMIN,
    SUPERADMIN,
}
const UserRole = Role.USER;
console.log(UserRole);

```

အလိုအလျောက် 0 ကနေ အစဉ်လိုက်စီထားတဲ့ တန်ဖိုးတစ်ခုကို ရသွားမှာပဲဖြစ်ပါတယ်။

လက်တွေ့မှာ ကိုယ်သတ်မှတ်ချင်တဲ့ တန်ဖိုးကိုပဲ အသုံးပြုကြတာများတယ်။

ဒို့အတွက် နောက်ပုံက Code အတိုင်း ကိုယ်ပိုင် တန်ဖိုးတွေ ထည့်သွင်းလုပ်ဆောင်နိုင်ပါ
တယ်။



သတ်မှတ်ထားတဲ့ တန်ဖိုးကို = နဲ့ Assign လုပ်ပြီး ကိုယ်ပိုင်တန်ဖိုးတွေကို အသုံးပြုလို့ရ ပါတယ်။

```

enum Permission {
    CREATE = "CREATE",
    READ = "READ",
    UPDATE = "UPDATE",
    DELETE = "DELETE",
}
const readPermission = Permission.READ;
console.log(readPermission);
  
```

ဒီလိုနည်းနဲ့ Enum ကို အသုံးပြုပြီး Constant Data အစာမျက်နှာ အဖြစ်အသုံးပြုကြလေ့ရှိ ပါတယ်။

တရေးသူကိုယ်တိုင်ကတော့ Enum Type ကို သိပ်သုံးလေ့သုံးထမ္မာပါဘူး။



TS - Array

```
let objectData : type [] = [type, type];
```

Type

let data: number[] = [1, 2, 3]; ရေးထုံးနဲ့ Number Type Array ကို ရေးသားနိုင်ပါတယ်။

လေးထောင့်ကွင်း အဖွင့်အပိတ် [] ရဲ့အရေးမှာ Type တန်ဖိုးကို သတ်မှတ်ပေးရိုပါ။

```
let objectData : type [] = [type, type];
let stringArr : string [] = [ "Hello", "World" ];
let numberArr : number [] = [ 100, 2027 ];
let booleanArr : boolean [] = [ true, false ];
```

Type

မိမိစက်ထဲမှာ type တွေကြောင်းလဲ သတ်မှတ်ပြီး ရေးသာနိုင်ပါတယ်။

ထူးခြားသွားမှာက လက်ခံရလိုက်တဲ့ variable နောက်မှာ dot (.) ရိုက်လိုက်တာနဲ့ TypeScript Server ကနေ push, shift အစရှိတဲ့ array methods တွေကို Feedback ပေးလာမှာဖြစ်ပါတယ်။





Type

```
let objectData : Array<type> = [type, type];
let stringArr : Array<string> = ["Hello", "World"];
let numberArr : Array<number> = [100, 2027];
let booleanArr : Array<boolean> = [true, false];
```

နောက်တစ်နည်းနဲ့ ရေးသားနိုင်ပါသေးတယ်။

Array ရဲ့ နောက်မှာ Less-than Sign < နဲ့ Greater-than Sign > ကြေားမှာ Type
ထည့်သွင်း ရေးသားနည်းပါ။

TS - Object

Type

```
let objectData : { name: string } = { name: "Lwin Moe Paing" };
```

အလယ်မှာ ":" သုံးပါတယ်

ရုံးရိုး JavaScript မှာလိုပဲ Object တစ်ခု Key-Value Pairs တွေနဲ့ တည်ဆောက်ပါတယ်

Object Type ကို ရေးသား ဖို့အတွက် TypeScript မှာ Property နဲ့ ValueType ကို
သတ်မှတ်ရေးသားကြပါတယ်။

Object Type Annotation

```
const person : {  
    name : string;  
    age : number;  
    status : "active" | "inactive";  
} = {  
    name : "Lwin",  
    age : 28,  
    status : "active",  
};
```

Type

Assigned
Value

const person ရဲ့ နောက်မှာ object type အဖြစ် ဆိုင်ရာ property တွေနဲ့ type တွေ အရင်သတ်မှတ်လိုက်ပါတယ်။

ပြီးတဲ့ နောက်မှာ တန်ဖိုးကို Assign လုပ်လိုက်ပါတယ်။

အောက်က Code အတိုင်းဖြစ်သွားမှပါ။

```
const person: {  
    name: string;  
    age: number;  
    status: "active" | "inactive";  
} = {  
    name: "Lwin",  
    age: 28,  
    status: "active",  
};
```



တာဖတ်သူတို့က property အသစ် isAdmin (boolean) type ထည့်ပေးပါ။

အဖွဲ့ကတော့ အောက်ပါ Code အတိုင်း ရလာမှာဖြစ်ပါတယ်။

```
const person: {
    name: string;
    age: number;
    status: "active" | "inactive";
    isAdmin: boolean
} = {
    name: "Lwin",
    age: 28,
    status: "active",
    isAdmin: true
};
```

Optional Properties

```
const person : {
    name : string;
    age ?: number;
    status ?: "active" | "inactive";
} = {
    name :"Lwin",
};
```

?: သုံးပြီး Optional Properties လုပ်လို့ရပါတယ်။

Object ထဲက တချို့ property တွေ မဖြည့်လည်း ရအောင် ? (question mark) ကို သုံးနိုင်ပါတယ်။

နှုန္နဘ code ထဲမှာ age , status ရယ်ကို ?: သုံးပြီး ရေးသားလိုက်တဲ့အတွက် Assign မထည့်သဲ ရေးသားနိုင် သွားပါတယ်။

TS - Type keyword

```
type CustomType = type;
```

“=” ဖြင့် Type တန်ဖိုးထည့်သည်။

TypeScript မှာ type keyword ကို Custom Type ပြုလုပ်ဖို့ အသုံးပြုပါတယ်။

```
type StringOrNumber = string | number;
const str : StringOrNumber = "Lwin";
```

ဒီ နမူနာ Code ထဲမှာ StringOrNumber ဆိုတဲ့ Custom Type တစ်ခုပြုလုပ်ပါတယ်။

နောက် constant variable str ကို annotation လိုက်ပါတယ်။

ဒုက္ခကြာင့် string type နဲ့ number type ကို လက်ခံလိုရသွားပါတယ်။

လက်တွေ့မှာတော့ Object Type တွေကို Custom Type တည်ဆောက်ပြီး အသုံးပြုတာ များပါတယ်။

ပိုပြီးရှင်းလင်းသွားသလို । ခကေခကာ ပြန်လည် အသုံးပြုချိန် (Reusable) ဖြစ်သွားတာ ကြောင့် ဖြစ်ပါတယ်။



```

type PersonType = {
    name : string;
    age : number;
    status : "active" | "inactive";
}

const personOne : PersonType = {
    name : "Lwin",
    age : 28,
    status : "active",
};

const personTwo : PersonType = {
    name : "Moe",
    age : 28,
    status : "inactive",
};

```

Person Type

နှောက် Code အရ type ကို အသုံးပြုပြီး PersonType တစ်ခု လုပ်လိုက်ပါတယ်။

အထဲမှာ Object Type ဖြစ်တဲ့အတွက် သက်ဆိုင်တဲ့ property နဲ့ တန်ဖိုး Type တွေကို သတ်မှတ်ထားပါတယ်။

ပြီးတဲ့နောက် constant variable နှစ်ခုဖြစ်တဲ့ personOne နဲ့ personTwo အတွက် PersonType တစ်ခုတည်းကိုပဲ အသုံးပြုနိုင်သွားပါပြီ။



TS - Tuple

Turple

```
const tupleData : [ type, type, ...type ] = [ value, value, ...value];
```

TypeScript မှာ Tuple ဆိုတာ array တစ်မျိုးပြုစီပါတယ်။

အထူးသေဖြင့် ကွဲပြားတဲ့ data type တွေပါဝင်ပြီး၊ Fix Length အဖြစ် သတ်မှတ်ရပါတယ်။

အရင်ဆုံး Color Code တွေသတ်မှတ်လို့ရမယ့် RGB (Red, Green, Blue) Color Tuple လေး ပြုလုပ်ကြည့်ရအောင်...

```
type ColorTuple : [ number, number, number ];
```

```
const redColor : ColorTuple = [ 255, 0, 0 ];
```

RGB တန်ဖိုးအတွက် Fix Length (၃) နေရာပဲ ယူလိုက်ပါမယ်။

တန်ဖိုး Type တွေကိုလည်း number ပဲ ထည့်လိုက်ပြီး Color Tuple Type အနေနဲ့ Custom Type သတ်မှတ်ထားတယ်။

ပြီးနောက် redColor variable ကို ColorTuple အဖြစ် တန်ဖိုး Assign ထည့်ပါမယ်။

ခုက္ခက သတ်မှတ်ထားတဲ့အတိုင်း Array Length (၃) နေရာပဲ number type တန်ဖိုးကို ထည့်သွင်းနိုင်မှာပါ။

အခါး တာဖတ်သူက Color Tuple အဖြစ်ဖန်တီးလိုက်နိုင်ပါပြီ။



```
const redColor : ColorTuple = [ 255, 0, 0 ];

const [ red, green, blue ] = redColor;
```

Tuple ကို Array Destructuring နည်းလမ်းနဲ့ ယူပြီး Variable တွေကို ပြုလုပ်နိုင်ပါ တယ်။

နူးနာ Code အရ redColor[0] တန်ဖိုးက အနီရောင် တန်ဖိုးဖြစ်မှာပါ။

Destructuring ရေးသားနည်းနဲ့ index တစ်ခုချင်းထိကို constant variable သုံးပြီး၊ red, green, blue variables တွေ တည်ဆောက် အသုံးပြုနိုင်ပါတယ်။

```
type StrBoolNum : [ string, boolean, number ];

const randomTuple : StrBoolNum = [ "Hello", true, 100 ];

const [ str, boo, no ] = randomTuple;
```

ဒီတခါ မတူညီတဲ့ Data Type (၃) မျိုးကို ထည့်ပြီး StrBoolNum Tuple လေး တည်ဆောက်ထားပါတယ်။

မိမိ စက်တဲ့မှာ စမ်းသပ်ရေးသား ကြည့်ပြီး Console ထုတ်ကြည့်ပါ။

အကြခံအားဖြင့် Tuple Type ဆိုတာ ဘာလဲ နားလည်လောက်ပါပြီ။



React useState hook tuple

လက်တွေ့မှာ အသုံးပြုတဲ့ Tuple Example လေးတစ်ခု ဖော်ပြလိုက်ရပါတယ်။

စာဖတ်သူတို့က React ကို လေ့လာဖူးတယ်ဆိုရင် React မှာ သုံးထားတဲ့ useState hook ရဲ့ Return type က Tuple ဖြစ်နေတာကို သိနိုင်မှာပါ။

```
type UseStateReturnType<T> = [ T, (val: T) => void ];
const useState = <T>(val: T) : UseStateReturnType<T> => [ val, (val: T) => {} ];
const [ val, setValue ] = useState("Hello");
```

useState hook ကိုခေါ်လိုက်တဲ့အခါ ကိုယ်ထည့်လိုက်တဲ့ Type တန်ဖိုးနဲ့ function type ကို Array အနေနဲ့ပြန်ရလာမှာဖြစ်ပါတယ်။ ဒါဟာ Tuple ပြဖြစ်ပါတယ်။

TS - Function

Function Type

```
type Func = ( param: type ) => type;
```

function type တစ်ခုကို ပြုလုပ်မယ်ဆိုရင် Parameter type ရယ်၊ Return type ရယ် သတ်မှတ်ပေးဖို့လိုပါတယ်။ Parameter မပါဘူးဆိုရင် () empty parameter အနေနဲ့ ရေးသားလိုလည်းရပါတယ်။ အရင်ဆုံး addFn ဆိုတဲ့ ဂဏန်း (J) လုံးပေါင်းတဲ့ function တစ်ခု တည်ဆောက်ပါမယ်။

Arrow Function (=>)

```
ts index.ts X
  ts index.ts > ...
  1
  2
  3 ✓ const addFn = (a: number, b: number): number => {
  4   |   return a + b;
  5 };
  6
  7
  8 console.log(addFn(1, 2));
```

addFn arrow function အဖြစ် ရေးသာမှာဖြစ်ပါတယ်။

Parameters အနေနဲ့ a ရယ် b ရယ် အတွက် number type တွေ လက်ခံထားပါတယ်။

ဦးတဲ့နောက် : နောက်မှာ Return Type အဖြစ် Number Type ပြန်မယ်လို့ ရေးသားလိုက်ပါတယ်။

အခု ဆို Function တစ်ခု ဘယ်လို ဖန်တီး ရမလဲဆိုတာ လေ့လာခဲ့ပြီး ပါပြီ။

ဒါဆို Manual သတ်မှတ်ထားတဲ့ Return Type ကို ဖြုတ်ကြည့်လိုက်ပါမယ်။

```
ts index.ts X
  ts index.ts > ...
  1 const addFn = (a: number, b: number) => {
  2   |   return a + b;
  3 };
  4
  5
  6
  7
  8
  9 const result = addFn(1, 2);
```

Return Type ကို Inference ပုံစံ TypeScript က အလိုအလျောက် သတ်မှတ်ပေးသွားမှာပါ။



Normal Function

Arrow Function ရေးသားနည်း လေ့လာပြီးသွားတဲ့အခါ ရှိုးရိုး function keyword နဲ့ ရေးသားထားတာကို ကြည့်ကြအောင် ... !!

```
index.ts
1 index.ts > ...
2
3 function addFn(a: number, b: number): number {
4     return a + b;
5 }
6
7
8 const result = addFn(1, 2);
```

- Parameter တိုင်းတို့ Type
သတ်မှတ်ပေးရပါမယ် !!
- ":" ရှိုးရိုး
- Return Type ပါ။

= နဲ့ Arrow => မပါတာက လွှဲရင် ကျွန်ုတ္တုပုံစံကတော့ အတူတူပါပဲ !!

အနောက်မှာလည်း Return Type ကို Manual သတ်မှတ်ပေးနိုင်ပါတယ် !!

```
index.ts
1 index.ts > ...
2
3 function addFn(a: number, b: number) {
4     return a + b;
5 }
6
7
8 const result = addFn(1, 2);
```

- Parameter တိုင်းတို့ Type
သတ်မှတ်ပေးရပါမယ် !!
- Return Type သတ်မှတ်ပေးပါ။
- အလိုအလောက် Number Type အဖြစ်
Return ပြန်သည်။

Return Type ကို ဖြတ်ကြည့်ပြီး Inference လုပ်သွားတာကိုလည်း အခုလို ရေးသားနိုင် ပါတယ် !!

Custom Type Function

type keyword နဲ့ Custom Function Type ရေးသားနည်းကို လွှဲလာသွားကြပါမယ်။

```

index.ts U
index.ts > ...
1  type AddFn = (a: number, b: number) => number;
2
3
4
5  const addFn: AddFn = (a, b) => {
6    return a + b;
7  };
8
9
10 const result = addFn(1, 2);
11
12
13
14

```

Custom Function Type
သတ်မှတ်တဲ့ ပုံစံပါ

" => " အနောက်မှာ Return Type
ဖြစ်ပါတယ်။

ထူးခြားချက်အနေနဲ့ : မဟုတ်ဘေးပဲ => arrow sign နဲ့ Return Type သတ်မှတ်ပေး
တာ သတိပြုရမှာပါ။ ကျွန်တာကတော့ အတူတူပဲဖြစ်ပါတယ်။ AddFn Type ကို
variable ကယူသုံးတဲ့အခါ parameter a နဲ့ b ကို type တွေ ထပ်သတ်မှတ်ပေးစရာမလို
တော့ပါဘူး။ AddFn Type ထဲမှာတင် အနဲ့ b ကို number type ဆိုတာကို TypeScript
က ကြိုတင်သိနေပြီးသားမို့ဂို့ဖြစ်ပါတယ်။

Function Void Type

```

index.ts U
index.ts > ...
1  const printFn = (message: string): void => {
2    console.log(message);
3  };
4
5
6
7  printFn("Hello World");
8
9

```

Return Type မပါဘူးအခါ void ဆိုပြီး
ရေးယုံပါပဲ။

Return တန်ဖိုးမရှိတဲ့ function တွေအတွက် void တန်ဖိုး ထည့်သွင်း အသုံးပြုနိုင်ပါ
တယ်။

Function Overloading

Function Overloading ဆိတာက function တစ်ခဲပဲ သုံးပြီး အမျိုးမျိုးသော Parameter type တွေ၊ Return type တွေကို သီးခြားသတ်မှတ်ဖို့ ဖြစ်ပါတယ်။

- Different Input Types ကို Handle လုပ်နိုင်ဖို့
- Code Readability & Maintainability တိုးစေဖို့
- Function name တစ်ခဲပဲ သုံးပြီး အမျိုးမျိုးသော လုပ်ဆောင်မှုများစွာ ဖန်တီးနိုင်ဖို့ ဖြစ်ပါတယ်။

ဥပမာ အနေနဲ့ Parameters (၂) မျိုးလက်ခံမယ့် add function ကို စဉ်းစားကြည့်ပါမယ်။ Idea က number နှစ်လုံးထည့်ရင် ပေါင်းပေးမယ်၊ string နှစ်လုံးထည့်ရင် string (၂) ခုကို concat လုပ်ပေးလိုက်မယ်။

Function Overloading ကို အသုံးပြုပြီး ရေးသားကြပါမယ်။

သူမှာ အပိုင်း (၂) ပိုင်းပါဝင်ပါတယ် ပထမအပိုင်းက Signature သတ်မှတ်ခြင်း၊ ဒုတိယ အပိုင်းက Code Implementation လုပ်ခြင်းပဲ ဖြစ်ပါတယ်။

```
add ( 10 , 10 )      =>      20
```

```
add ( "Hello, " , "World" ) . . . =>      "Hello, World"
```



Signature သတ်မှတ်ခြင်း

အရင်ဆုံး function keyword ကို သုံးပြီး ကိုယ်ဖြစ်စေချင်တဲ့ signature ကို ရေးသားရမှာ ဖြစ်ပါတယ်။

```
function add ( a : number , b : number ) . . . : number
```

```
function add ( a : string , b : string ) . . . : string
```

1st Signature မှာ parameter a number type ဖြစ်ပြီး parameter b လည်း number ဖြစ်ရင် number type ပဲ return ပြန်ပါမယ်။

2nd Signature မှာ parameter a string type ဖြစ်ပြီး parameter b လည်း string ဖြစ်ရင် string type ပဲ return ပြန်ပါမယ်။

ပြီးတဲ့အခါ သတ်မှတ်ထားတဲ့ Signature ရဲ့ Parameter တွေကို Cover ဖြစ်အောင် Code Implementation ပြန်လုပ်ပေးရမှာပါ။

Implementation ပြုလုပ်ခြင်း

```
function add ( a : number | string , b : number | string ) . . . : number | string {  
    // Code Implementation  
}
```

1st Signature မှာ a parameter သည် number type လက်ခံပြီး 2nd Signature အတွက် string type ကိုလက်ခံထားပါတယ်။ ဒါကြောင့် Implementation Function သည် signature ၂ ခုလုံး cover ဖြစ်အောင် ပထမ Parameter ကို number | string နဲ့ လက်ခံရပါမယ်။ parameter b အတွက်ကော် return type အတွက်ပါ cover ဖြစ်အောင် string | number နဲ့ ပြောင်းလဲပြီး implement လုပ်ထားလိုက်ပါပြီ။

ပြီးတဲ့အခါက္ခမှ JavaScript Code Implementation အဆင့်ရောက်ပြီး Logic ကို
ရေးသားကြရမှာဖြစ်ပါတယ်။

```

ts index.ts ×
index.ts > ...
1 // Function Overload
2
3 function add(a: number, b: number): number; -
4 function add(a: string, b: string): string; -
5 function add(a: number | string, b: number | string): number | string {
6   if (typeof a === "string" && typeof b === "string") {
7     return a.concat(b);
8   }
9
10  if (typeof a === "number" && typeof b === "number") {
11    return a + b;
12  }
13
14  throw new Error("Invalid arguments");
15
16
17 const resultNumber = add(10, 20); // Return 30
18
19 const resultString = add("Hello, ", "TS"); // Return "Hello, TS"
20

```

Code အပြည့်အစုံကို ရေးသားဖော်ပြလိုက်ပါတယ်။ ဒီလောက်ဆုံး Function Overload
ကို အကြော်အားဖြင့် နားလည်သွားပြီလို့ ယူဆပါတယ်။



TS - Intersection

Intersection

```
type TypeA = { name: string };
type TypeB = { age: number };

type CombinedType = TypeA & TypeB;
```

TypeScript မှာ Intersection (&) Operator ကို Type တွေကို ပေါင်းစပ် (combine) လုပ် ဖို့သုံးပါတယ်။ Type နှစ်ခု (သို့မဟုတ် ပိုများတဲ့ Type) ကို ပေါင်းစပ်ပြီး အားလုံးရဲ့ property တွေ ပါဝင်တဲ့ Type အသစ်တစ်ခု ဖန်တီးနိုင်ပါတယ်။

```
index.ts M ×
023-ts-intersection > index.ts > ...
You, 27 minutes ago | 1 author (You)
1 type People = {
2   name: string;
3   age: number;
4 };
5
6 type AdminRole = {
7   role: "admin";
8 };
9
10 type Admin = People & AdminRole; ●-----> Intersection Admin (&)
11
12 const admin: Admin = {
13   name: "LMP",
14   age: 20,
15   role: "admin",
16 };
17
18 console.log(admin);
19
```

Code ကို တစ်ချက် လေ့လာကြည့်ပေးပါ။

People Type နဲ့ AdminRole Type ကို (&) နဲ့ ပေါင်းစပ်လိုက်တာပဲ ဖြစ်ပါတယ်။

ဒါခို့ရင် Admin Type အသစ်မှာ properties အားလုံးကို လက်ခံရရှိသွားမှာပါ။



TS - Interface

Interface keyword

TypeScript Interface ကို interface keyword ကိုသုံးပြီး object type ကို ဖြေလှပနိုင်ပါတယ်။

```
interface User {
  id: string;
  name: string;
  age?: number;
  greet: () => void
}

const user: User = {
  id: "102";
  name: "Lwn Moe Pang";
  // `age` is optional
  greet: () => {}
}
```

Extending Interfaces

Extending Interfaces ဆိုတာ Interface တစ်ခုကနေ အသစ်တိုးချဲ့ပြီး အသုံးပြနိုင်တဲ့ နည်းလမ်း ဖြစ်ပါတယ်။

လေ့လာခဲ့တဲ့ Intersection နဲ့ နည်းနည်းဆင်ပါတယ် Interface နှစ်ခုပေါင်းစပ်လိုက်တဲ့ ပုံစံဖျိုးပါ။

```
index.ts U •
index.ts > ...
1 ↵ interface Person {
2   |   name: string;
3   |   age: number;
4 }
5 ↵ interface Employee extends Person {
6   |   role: string;
7 }
8
9
10 ↵ const employee: Employee = {
11   |   name: "Alice",
12   |   age: 30,
13   |   role: "Developer",
14 };
15
```

Employee \cap Person ကို
extends လုပ်လှပတော်ဗူး
Person ရဲ့Properties တွေကို
ရရှိလားပါတယ်



Interface As Function

interface keyword နဲ့ Function Type တစ်ခုအနေနဲ့ အခုလိုရေးသားအသံးပြနိုင်ပါတယ်။

```

index.ts U X
index.ts > ...
1
2  interface PrintFn {
3    (str: string): void;
4  }
5
6
7 const print: PrintFn = (log) => {
8   console.log(log);
9 };
10

```

TS - readonly

```

type MyType = {
  readonly property: string;
}
let noArray : readonly number[] = [ 1, 2 ]

```

TypeScript မှာ readonly ဟာ property (ဘို့) array ကို read-only (ဖတ်လိုရပဲရအောင်) သတ်မှတ်နိုင်တဲ့ keyword ဖြစ်ပါတယ်။

မှားယွင်းပြီး ထည့်မိတာမျိုး ပြောင်းလဲတာမျိုးကို ကာကွယ်ချင်တဲ့အခါ သုံးကြပါတယ်။
Modify လုပ်လို့မရပါဘူး။

Readonly property

```

ts index.ts 1, U •
ts index.ts > ...
1 type Person = {
2   readonly id: number;
3   name: string;
4 };
5
6 const user: Person = {
7   id: 1,
8   name: "Lwin",
9 };
10
11 user.name = "Lwin Moe Paing"; ⚡
12
13 Cannot assign to 'id' because it is a read-only property. ts(2540)
14 △ Error (TS2540) ⚡ | ⓘ
15
16 Cannot assign to id because it is a read-only property.
17
18 (property) id: any
19
20 View Problem (F8) No quick fixes available
21 user.id = 2; ⚡

```

readonly id နဲ့ Person Type
ပြလုပ်ပါမည်။

constant user ပြလုပ်ပါမည်။

property တို့ Assign လုပ်နိုင်ပါသည်

read-only တို့ Assign မလုပ်နိုင်ပါ

Person Type မှာ id property ကို readonly keyword နဲ့ ထိန်းလိုက်တာပါ။

ဒါခိုရင် id ကို အခြား value တွေနဲ့ modify (ပြုပြင်) လို့ မရတော့ပါဘူး။
အခြား name property ကိုတော့ ပြောင်းလဲလို့ ရနိုင်ပါတယ်။

Readonly Array

TypeScript မှာ array တွေကိုလည်း readonly keyword နဲ့ push, pop, shift, unshift method တွေကို သုံးလို့မရအောင် ထိန်းနိုင်ပါတယ်။

```

ts index.ts ./ 1, U • ts index.ts 029-ts-readonly
ts index.ts > ...
1 const nums: readonly number[] = [1, 2, 3];
2
3 Property 'push' does not exist on type 'readonly number[]'. ts(2339)
4 △ Error (TS2339) ⚡ | ⓘ
5
6 Property push does not exist on type readonly number[] .
7
8 any
9
10 View Problem (F8) No quick fixes available
11 nums.push(2);

```



TS - keyof

```
type Keys = keyof ObjectType
```

TypeScript မှာ keyof က Object တစ်ခုရဲ့ Keys (Property Names) တွေကို Union Type တစ်ခုအဖြစ်ပြန်ထုတ်ပေးနိုင်ပါတယ်။ keyof ObjectType ရေးထံးနဲ့ String Union တွေကို ရရှိသွားမှာပါ။

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

```
type UserKeys = keyof User
"id" | "name" | "age"
```

နှေ့နာ Code ထဲမှာ id, name, age အစရှိတဲ့ property တွေကို keyof User နဲ့ ထုတ်ယူလိုက်နိုင်ပါတယ်။

```
index.ts 1, U •
index.ts > [o] oneOfUserKey
1 type User = {
2   id: string;
3   name: string;
4   age: number;
5 };
6
7 type UserKeys = keyof User;
8
9 const oneOfUserKey: UserKeys = ""
10
11
12
```

TypeScript server က လည်း "id" | "age" | "name" တွေကို Feedback ပေးသွားမှာဖြစ်ပါတယ်။

အခု ရေးသားနည်းအရ UserKeys Type တစ်ခု ခွဲထဲတ်ရေးသားထားခြင်းဖြစ်ပါတယ်။

တိုက်ရှိက် keyof User ပုံစံနဲည်း အသုံးပြန်စိတ်တာကို လေ့လာကြည့်ပါမယ်။

```

ts index.ts 1, U •
ts index.ts > [e] oneOfUserKey
1 type User = {
2   id: string;
3   name: string;
4   age: number;
5 };
6
7 const oneOfUserKey: keyof User = "";
8

```

The screenshot shows a code editor with a tooltip for the variable 'oneOfUserKey'. The tooltip lists the three keys of the 'User' type: 'age', 'id', and 'name'. The 'age' option is highlighted.

type အသစ် မကြောင်းတော့ပဲ တိုက်ရှိက် constant oneOfUserKeys အနောက်မှာ keyof syntax နဲ့ ရေးသားလိုက်တာကို မြင်ရမှာပါ။ keyof အကြောင်းအရာက ဒီလောက်ပဲ ဖြစ်ပါတယ်။

TS - typeof

typeof

typeof ကို JavaScript runtime မှာ သုံးမယ်ဆိုရင် variable ရဲ့ type ကို string အနေနဲ့ return ပြန်ပါမယ်။

TypeScript ရေးသားနည်းမှာ typeof ကို သုံးမယ်ဆိုရင်တော့ variable ရဲ့ Type Structure ကို ပြန်ရမှာဖြစ်ပါတယ်။



```
const user = {
    name: "Lwin Moe Paing";
}
```

Constant Object Variable
တစ်ခုပဲဖြစ်ပါတယ်။

TS

```
type UserType = typeof user
```

```
{
    name: string;
}
```

JS

```
console.log(typeof user)
```

"object"

အခါ နမူနာ Code တဲ့ကအတိုင်း typeof user က JavaScript Runtime မှာဆို “object” ဆိုတဲ့ String ကိုရသွားမှာပါ။

TypeScript ရေးသားနည်းမှာ typeof variable ပုံစံနဲ့ Object Structure ကို Custom-Type အဖြစ် ပြုလုပ်နိုင်ပါတယ်။

appConfig Code ကို လေ့လာသွားကြပါမယ်။

ပထမဆုံး appName နဲ့ env property တွေပါတဲ့ appConfig ဆိုတဲ့ constant variable တည်ဆောက်ထားပါတယ်။



```

index.ts U X
index.ts > ...
1 const appConfig = {
2   appName: "MyCoolApp",
3   env: "default",
4 };
5
6 type ConfigType = {
7   appName: string;
8   env: string;
9 }
10 type ConfigType = typeof appConfig;
11 type ConfigType = typeof appConfig;
12
13
14 const devConfig: ConfigType = {
15   appName: "MyCoolAppDev",
16   env: "dev",
17 };
18
19 const prodConfig: ConfigType = {
20   appName: "MyCoolAppProd",
21   env: "prod",
22 };

```

Constant appConfig
Variable ရဲ type စု Alias
ယူလိုက်တဲ့ဖြစ်ပါတယ်။

Custom Type တစ်ခွာအဖြစ် ConfigType ကို constant appConfig က တဆင့် typeof appConfig သုံးပြီး ပြုလုပ်လိုက်ပါတယ်။

အေား devConfig နဲ့ prodConfig Config Variables တွေအတွက် Reusable Type အဖြစ် အသုံးပြနိုင်သွားပါပြီ။

Type တွေကို ကိုယ်တိုင် သတ်မှတ်တာမဟုတ်ဘဲ Variable ဆိုက Type ကို ပုံတူဗျားတာ မျိုး ဖြစ်တဲ့အတွက် အသုံးများပါတယ်။

```

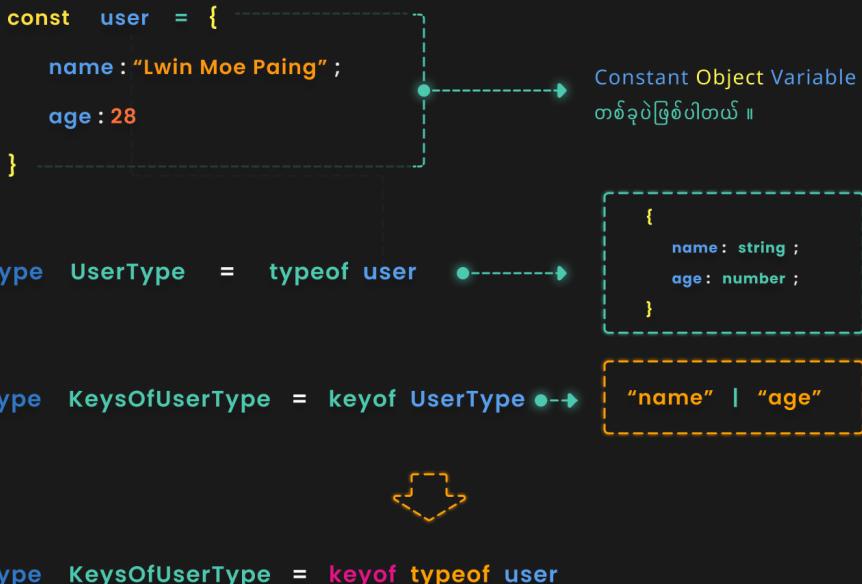
index.ts U X
index.ts > ...
1 const appConfig = {
2   appName: "MyCoolApp",
3   env: "default",
4 };
5
6 const devConfig: typeof appConfig = {
7   appName: "MyCoolAppDev",
8   env: "dev",
9 };
10
11 const prodConfig: typeof appConfig = {
12   appName: "MyCoolAppProd",
13   env: "prod",
14 };

```

Constant appConfig
Variable ရဲ type စု Alias
ယူလိုက်တဲ့ဖြစ်ပါတယ်။

Custom Type မတည်ဆောက်တော့ပဲ တိုက်ရှိက် variable: `typeof otherVariable` အခါ
လိုပုံစံမျိုးနဲ့လည်း ရေးသားတာမျိုး ရှိပါတယ်။ ဒါက ရှိုးရှင်းတဲ့ Type Alias တစ်မျိုးပဲ
ဖြစ်ပါတယ်။

`keyof typeof` တွဲရေးခြင်း



ဒါကလည်း အသုံးများတဲ့ ပုံစံလေးဖြစ်ပါတယ်။

`typeof` ကို သုံးလိုက်တဲ့အခါ variable ရဲ့ Type Structure ကိုရသွားမှပါ။

ပြီးတော့မှ သူ့ရဲ့ အရေ့မှာ `keyof` နဲ့ `union keys` တွေကို ထုတ်ယူလိုက်တာပဲဖြစ်ပါ
တယ်။



TS - Index Type

```
type ObjectType = {
  [ index : KeyType ] : ValueType;
}

[ index Name ]           [ string | number | symbol ]
```

Index Signature

TypeScript မှာ Index Signature ဆိတာက object properties တွေကို dynamic key အနေဖြင့် သုတေသနပေးတဲ့ concept တစ်ခုပဲဖြစ်ပါတယ်။ ရေးသားနည်း [index: KeyType] က property type သတ်မှတ်တာဖြစ်ပြီး (:) အနေကိုမှာ value type တွေကို သတ်မှတ်တာပဲဖြစ်ပါတယ်။

```
index.ts
1 type NumberObj = {
2   [index: number]: string;
3 }
4
5 const productNames: NumberObj = {
6   1: "Laptop",
7   2: "Phone",
8   3: "Tablet",
9 }
10
11 console.log(productNames[1]);
12 // Result => Laptop
13
```

Property Index ကို နံပါတ်တွေပဲထားစေလို့ သတ်မှတ်တာပါ

နူးနာ code အရ [index: number]: string ရေးသားထားတာက keys တွေအားလုံး number type ဖြစ်ပြီး value တွေက string type ဖြစ်မယ်လို့ သတ်မှတ်ထားတာပါ။

စာဖတ်သူတို့က key ကို string type ထားပြီး value ကို boolean type အဖြစ် မိမိ စက်ထဲမှာ ရေးသားကြည့်ပေးပါ။

အဖောက် အောက်က Code အတိုင်း ဖြစ်မှာပါ။

```
type StrBoolObj = {
    [index: string]: boolean;
}
```

TS - Predefined & Index

```
type ObjectType = {
    [ index : KeyType ] : ValueType;
}
[ predefined ] -- [ key1 : ValueType;
key2 : ValueType;
]
```

TypeScript မှာ Index Signature ကိုသုံးပြီး Object တစ်ခုအတွင်း predefined properties ထွေကိုသတ်မှတ်နိုင်ပါတယ်။

```
index.ts ✘
index.ts > ...
1  type StringObj = {
2      [index: string]: string;
3      name: string;
4      address: string; }-- [ predefined ]
5  };
6
7  const user: StringObj = {
8      name: "Lwin",
9      address: "Ygn",
10     // Other Properties
11     other: "values",
12 };
13
14 console.log(user.name);
15
16
```

StringObj ထဲမှာ KeyType string ရယ် ValueType string ဖြစ်တဲ့အပြင်။

name property နဲ့ address property ပါကိုပါရမယ်လို့ Predefined လုပ်ထားတာဖြစ်ပါတယ်။

TS - Union keys in Index

```
type ObjectType = {
    [key in UnionType] : ValueType;
}
```

TypeScript မှာ Union Type () ကို အသုံးပြုပြီး Object Type ကို ပြုလုပ်နိုင်ပါတယ်။

Syntax အရ [key in UnionType] : ValueType ရေးသားနည်းပါ။



```
index.ts > ...
1  type UserKeys = "name" | "age" | "email";
2
3  type User = {
4      [key in UserKeys]: string;
5  };
6
7  const user: User = {
8      name: "John",
9      age: "25",
10     email: "john@example.com",
11 };
12
13
```

The code defines a type `UserKeys` as a union of three strings: "name", "age", and "email". It then defines a type `User` as an object where each key is of type `UserKeys`. A dashed box labeled "Union" encloses the union type `"name" | "age" | "email"`.

နှေ့မှန်ဘူး Code အရ "name" | "age" | "email" ဆိတဲ့ `UserKeys` Union Type ရှိပါမယ်။

`[key in UserKeys]` ကို အသုံးပြုပြီး Union ထဲမှာပဲ ပါတဲ့ တန်ဖိုးတွေကို Key အဖြစ် သုံးလိုက်တာပါ။

အခုလို ရေးသားနည်းနဲ့ Object တွေကို Dynamic ဖော်တိုး ယူလို့ရပါတယ်။



TS - Generic

Generic ဆိတာ ?

< T >

Generic ဆိတာ အလွယ်အားဖြင့် မသိသေးတဲ့ data type ကို လက်ခံရနိုင်း အဲ data type ကို ပြန်လည်အသုံးချတဏဖြစ်ပါတယ်။ Syntax က < T > ကြားထဲမှာ Generic Type ကိုထည့်သွင်း အသုံးပြုတဏဖြစ်ပါတယ်။

Generic Function

<T> (value: T): T ရေးသားနည်းနဲ့ Generic Function ကို ရေးသားပါတယ်။

T က Dynamic Type ကို ရည်ညွှန်းထားတာပါ။

```
function identity <T> ( value : T ) : T {  
    return value ;  
}  
  
const identity = <T> ( value : T ) : T => {  
    return value ;  
}
```

T ထဲကို အမြား Type တွေကို အစားထိုး ကြည့်မယ်ဆိုရင် ...



`< T > (value : T) : T`

`T = string` `< string > (value : string) : string`

`T = number` `< number > (value : number) : number`

`T = boolean` `< boolean > (value : boolean) : boolean`

T က string အနေနဲ့ အတားထိုးမယ်ဆိုရင်၊ လက်ခံမယ့် Parameter (value: string) ဖြစ်ပါး Return type က လည်း string ဖြစ်သွားမှာပါ။

T က number ဖြစ်မယ်ဆိုရင်၊ Parameter (value: number) ဖြစ်ပါး Return type က number ဖြစ်သွားမှာပါ။

T = boolean ဆိုလည်း ထိုနည်းလည်းကောင်းပါပဲ။

```
index.ts ✘
index.ts > ...
1 const identity = <T>(value: T): T => {
2   return value;
3 };
4
5 let str = identity("Hello World");
6
7   let str: string
8 console.log(str);
9 |
10
```

```
index.ts > ...
1 const identity = <T>(value: T): T => {
2   return value;
3 };
4
5 let num = identity(42);
6
7   let num: number
8 console.log(num);
9 |
10
```



ဘယ်ဘက် code အရ identity("Hello World") function ခေါ်လိုက်တာက । Parameter အရ string type ဖြစ်တဲ့အတွက် Return Type သည် string type ပဲ ရလာမှာဖြစ်ပါတယ် ။ ဒါကို str variable ကို mouse နဲ့ ခကာအကြား hover လုပ်လိုက်ရင် let str: string ဆိုပြီး ပေါ်လာတာတွေ နှင့်မှာပါ ။

ညာဘက်ပံ့ code identity(42) မှာ । Parameter value type ဖြစ်ပြီး Return ကလည်း value type ဖြစ်လာပါတယ် ။

ဒီနည်းလမ်းနဲ့ Generic Type <T> Function တွေ ရေးသားနှင့်ပါတယ် ။

အခါ Primitive Type တွေ အစား Complex ဖြစ်တဲ့ User Object တစ်ခုလုပ်ပြီး ထည့်သွင်း အသုံးပြုကြည့်ပါမယ် ။

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

< T > (value : T) : T

T = User

< User > (value : User) : User

User Object type တစ်ခု ဖန်တီးလိုက်ပါမယ် ။

စာဖတ်သူတို့က < T > နေရာမှာ User Type ကို အစားထိုးကြည့်လိုက်ပါ ။

Return Type ကလည်း User Type ရသွားမှာဖြစ်ပါတယ် ။



```

index.ts U X
index.ts > ...
1  const identity = <T>(value: T): T => {
2    return value;
3  };
4
5  type User = {
6    id: string;
7    name: string;
8    age: number;
9  };
10
11 const user: User = { id: "1", name: "LMP", age: 28 };
12
13 const result = identity(user);
14
15 // const result: User
16 console.log(result);
17

```

Code အပြည့်စုံကို မိမိစက်တဲ့မှာ ထည့်သွင်း စမ်းရေးကြည့်ပါ။

Exercise for Generic

Exercise အနေနဲ့ ဘဏ်တံ့သူတို့ကို ပုံစံတစ်ပုဒ် စဉ်းစားခိုင်းပါမယ်။

`successResponse("Hello")` function ကို ခေါ်သုံးလိုက်လျှင်၊ Return အနေနဲ့

`{ success: true, data: "Hello" }` ကို ပြန်ရမှာဖြစ်ပါတယ်။

`successResponse({ id: 1, name: "LMP" })` function မှာ၊ Return Type က

`{ success: true, data: { id: 1, name: "LMP" } }` ကို ပြန်ရမှာဖြစ်ပါတယ်။

ဘဏ်တံ့သူတို့က `successResponse` ကို generic function တစ်ခုအနေနဲ့ ရေးကြည့်ပါ။



```
successResponse ( "Hello" )
```

```
{
  success : true ;
  data : "Hello" ;
}
```

```
successResponse ( { id: 1 , name: "LMP" } )
```

```
{
  success : true ;
  data : {
    id : 1 ;
    name : "LMP" ;
  }
}
```

ဘုတ်ပါတယ် Return type နေရာမှာ { success: boolean, data: T } လိုပြောင်း
ပေးလိုက်ရင် မှန်ပါတယ်။ T နေရာမှာ အစားထိုးလိုက်တဲ့ Code ပုံလေးကို တစ်ချက်
ကြည့်ကြပါမယ်။

```
< T > ( data : T ) : { success : boolean , data: T }
```

T = string	< string >(value : string) : { success : boolean , data: string }
------------	---

T = number	< number >(value : number) : { success : boolean , data: number }
------------	---

T = User	< User >(value : User) : { success : boolean , data: User }
----------	---





```
index.ts U X
index.ts > ...
1  const successResponse = <T>(data: T): { success: boolean; data: T } => {
2    return {
3      success: true,
4      data: data,
5    };
6  };
7
8
9  let helloResult = successResponse("Hello");
10
11
12
13
14
15  console.log(helloResult);
16
```

helloResult ကို mouse နဲ့ ခကာအတွက် တင်ထားလိုက်ရင်။

return type ကို ထည့်လိုက်တဲ့ Type အတိုင်း Dynamic Return ပြန်နေတာတွေရမှာပါ။

Generic Type

type Generic<T> = T ရေးသားနည်းနဲ့ Generic Type ကို ရေးသားနိုင်ပါတယ်။

Exercise မှာရေးခဲ့တဲ့ Return Type ကိုပဲ Custom Generic Type အနေနဲ့ ရေးပြသွားပါမယ်။

<pre>type Response <T> = { success: boolean; data: T; }</pre>	<pre>T = string Response <string> = { success: boolean; data: string; }</pre>
---	---

Type ကို Response နာမည်နဲ့ Generic Type အမျိုးအစားရေးသားလိုက်ပါတယ်။

ညာဘက် Code က string type ကို အစားထိုး ရေးသားတာဖြစ်ပါတယ်။

```
ts index.ts ✘ ×
  ts index.ts > ...
● 1  ↵ type Response<T> = {
  2    success: boolean;
  3    data: T;
  4  };
  5
  6  ↵ const numberRes: Response<number> = {
  7    success: true,
  8    data: 10,
  9  };
 10
 11  console.log(numberRes);
 12
```

နဲ့နဲ့ Code ထဲကအတိုင်း Response ဆိုရင် data နေရာမှာ number type ဖြစ်မှာပါ။

<နဲ့> ကြားထဲမှာ Type တွေကို ပြောင်းလဲ ပြီး ပိမိစက်ထဲမှာ စမ်းနိုင်ပါတယ်။

```
ts index.ts ✘ ×
  ts index.ts > ...
  1  type Response<T> = {
  2    success: boolean;
  3    data: T;
  4  };
  5
  6  const successResponse = <T>(data: T): Response<T> => {
  7    return {
  8      success: true,
  9      data: data,
 10    };
 11  };
 12
 13  let helloResult = successResponse("Hello");
 14
```

နားလည်းခြားမြှုပ်နည်း ခုကာက Exercise ပုစ္စာရဲ့ Response Type ကို အချင်း
ပြောင်းလဲ ရေးသားလိုက်လို့ရပါပြီ။

TS - as Type Casting

```
let value = someVariable as TargetType;
```

TypeScript ရဲ့ “as” keyword ဟာ Type Assertion (Type Casting) လိုခေါပါတယ်။

someVariable as TargetType ရေးသားနည်းနဲ့ ပြောင်းလဲစေခဲ့တဲ့ Type ကို Modify လုပ်နိုင်ပါတယ်။

JSON parse နမူနာ Code ကို အရင်ကြည့်ရအောင် ...။

```
index.ts U •
index.ts > ...
1  type User = {
2    id: number;
3    name: string;
4  };
5
6  const responseData = JSON.parse('{"id":1,"name":"LMP"}');
7
8  const user = responseData as User;
9
10 console.log(user);
11
12
```

JSON.parse method ၏ any type ပဲ ပြန်ပေးပါတယ်။

ဒါကြာင့် responseData ထဲမှာ any ဖြစ်နေမှပါ။

သို့ပေါ်မယ် မိမိက User Type အနေနဲ့ ပြန်ရမယ်ဆိတာ ကြိမ်းသေသိနေတဲ့ အခြေအနေ မျိုးဖြစ်နေတဲ့အပါ။

User Type အနေနဲ့ Type Casting လုပ်လိုက်တာ Type safe ပိုဖြစ်သွားမှာပါ။

နောက်ပိုင်း user variable ကို သုံးတဲ့အပါ TypeScript Feedback တွေ သေချာရနိုင် သွားမှာပါ။



```
ts index.ts U X
ts index.ts > ...
1 const anyData: any = "Something";
2
3 const strValue = anyData as string;
4
5 console.log(strValue);
6
7 const unknownData: unknown = 100;
8
9 const number = unknownData as number;
10
11         const number: number
12 console.log(number);
13
```

များသောအားဖြင့် ဘယ်အခြေအနေတွေမှာ Type Assertion နည်းကို အသုံးများလဲဆိုရင် ।

unknown type နဲ့ any type ရလာတဲ့ အခါမျိုးတွေမှာ

ကိုယ်ကလည်း ဘယ် Type လဲဆိုတာ သေချာသိနေတဲ့ အခြေအနေမှာ သုံးပါတယ် ။

TS - satisfies

```
let variable = value satisfies TargetType;
```

TypeScript မှာ satisfies operator က (၂) မျိုးလုပ်ဆောင်ပေးပါတယ် ။

အရင်ဆုံး Target Type နဲ့ Type-Checking လုပ်ပါတယ် ။

ပြီးတဲ့နောက် Type Safe ဖြစ်တဲ့ လုပ်ဆောင်ချက်ကို ထည့်ပေးပါတယ် ။

satisfies ရဲ့ Type-Checking လုပ်ဆောင်ပုံကို အရင်လေ့လာကြည့်ပါမယ် ။



```
type User = {
    id: string;
    strOrNo: string | number;
}
```

✓ const user = {
 id: "100";
 strOrNo: "Lwin Moe Paing";
} satisfies User;

✗ const user2 = {
 id: "100";
 age: 28;
} satisfies User;

constant variable ဖြစ်တဲ့ user ထဲကို တန်ဖိုးတွေ assign လုပ်ထားပါမယ်။

သူရဲ့ အနောက်မှာ satisfies operator TargetType ကို ကပ်ပြီးရေးထားတဲ့ ပုံစံဖြစ်ပါတယ်။

ဘယ်ဘက်ပုံအရ property id အတွက် တန်ဖိုး "100" ဖြစ်ပါတယ်။ UserType ဆုံးလည်း id: string ဖြစ်တာကြောင့် Type-Checking မှန်ပါတယ်။ ဒုတိယ property strOrNo: string | number မှာ strOrNo: "Lwin Moe Paing" မှိုလို့ Type-Checking မှန်ပါတယ်။ ဒါကြောင့် ဘယ်ဘက် Code မှာ Type Checking Pass ဖြစ်သွားပါတယ်။

ညာဘက် Code မှာ age ဆိုတဲ့ property က UserType ထဲမှာ မရှိပါဘူး။ ဒါကြောင့် Type-Checking က မှားနေကြောင့် TypeScript Server က Error ပြသွားမှာပါ။

satisfies operator ပဲ့ Type Safe ဖြစ်တဲ့ လုပ်ဆောင်ချက်ကို ထပ်လေ့လာသွားပါမယ်။



Satisfies

```
const user = {
    id: "100";
    strOrNo: "Lwin Moe Paing";
} satisfies User;
```

Type

```
const user : User = {
    id: "100";
    strOrNo: "Lwin Moe Paing";
};
```

String Suggestion

user.strOrNo.

```
charAt
charCodeAt
codePointAt
concat
endsWith
includes
etc...
```

String | Number Suggestion

user.strOrNo.

```
toLocaleString
toString
valueOf
```

ဘယ်ဘက် ပုံက User Type ကို satisfies နဲ့ အသုံးပြုထားပါတယ်။ သူက Type-Checking ပြီးသွားတဲ့ အခါ Type-Safe ဖြစ်အောင်ဆက်လုပ်ပါတယ်။ နမူနာ Code ထဲမှာ user.strOrNo ဆိုတဲ့ property value က "Lwin Moe Paing" ပါခြေကြောင့် string type ဆိုတာ သေချာသွားပြီလို့ ယူဆလိုက်ပါတယ်။ user.strOrNo(.) ဆိုပြီး ခက်ထားလိုက်တဲ့ အခါ string method တွေကိုပဲ TypeScript Server က feedback တွေပေးသွားမှာပါ။

ညာဘက်ပုံက User Type ကို annotation နည်းနဲ့ တန်ဖိုးကိုထည့်လိုက်ပါတယ်။ ser.strOrNo(.) ဆိုပြီး ခက်ထားလိုက်တဲ့ အခါ type က string | number ဖြစ်နေတဲ့ အတွက် string နဲ့ number မှာ ဘုံးတူတဲ့ method တွေကိုပဲ feedback ပေးသွားမှာပါ။

ဒါက ရိုးရိုး Type ကိုကြော်တာနဲ့ Satisfies ရဲ့ အသုံးပြုပဲ ကွာခြားချက်ပဲ ဖြစ်ပါတယ်။

Code ရေးသားပုံ အပြည့်အစုံကို လေ့လာကြည့်နိုင်ပါတယ်။

```
index.ts 1, U •
index.ts > ...
1 type User = {
2   id: string;
3   strOrNumber: string | number;
4 };
5
6 const user = {
7   id: "1",
8   strOrNumber: "Hello",
9 } satisfies User;
10
11   (property) strOrNumber: string
12
13 user.strOrNumber.
  ↴ Symbol
  ↴ charAt
  ↪ charCodeAt
  ↴ codePointAt
  ↴ concat
  ↴ endsWith
```

```
index.ts 1, U •
index.ts > ...
1 type User = {
2   id: string;
3   strOrNumber: string | number;
4 };
5
6 const user: User = {
7   id: "1",
8   strOrNumber: "Hello",
9 };
10
11   (property) strOrNumber: string | number
12
13 user.strOrNumber.
  ↴ toLocaleString
  ↪ toString
  ↴ valueOf
```

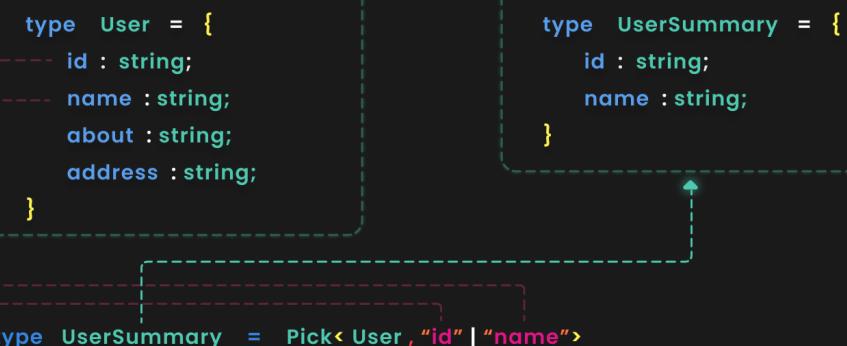


Utility

TS - Pick

```
type NewType = Pick<OriginalType, "key1" | "key2">
```

Pick Utility Type တ ရှိပြီးသား Object Type ထဲမှ ရွေးချယ်ချင်တဲ့ keys တွေနဲ့ Type အသစ်တစ်ခု ပြုလုပ်ပေးခြင်းလုပ်ဆောင်နိုင်ပါတယ်။



နဲ့နဲ့ Code ထဲမှာ User ဆိုတဲ့ type ထဲမှာ “id”, “name”, “about”, “address” ဆိုပြီး Property (င) ပျိုးရှိပါတယ်။

“id”, “name” ပဲ User Type ထဲက ယူချင်တယ်ဆိုရင် Pick ကို သုံးပြီး ရေးသားနိုင်ပါတယ်။

Pick<User, “id” | “name”> ရေးသားနည်းနဲ့ UserSummary ဆိုတဲ့ Type အသစ်ကို ဖန်တီးလိုက်တာပဲဖြစ်ပါတယ်။

TS - Omit

```
type NewType = Omit<OriginalType, "key1" | "key2">
```

Omit အတော့ Pick ရဲ့ ကြောင်းပြန်ပါပဲ။

Object Type ထဲကနေ ထုတ်ချင်တဲ့ Keys တွေကို ရွေးချယ်ပြီး Type အသစ်ပြုလုပ်ပေးနိုင်ပါတယ်။

```
type User = {
    id : string;
    name : string;
    about : string;
    address : string;
}

type UserSummary = {
    id : string;
    name : string;
}

type UserSummary = Omit<User, "about" | "address">
```

နမူနာ Code ထဲမှာ User ဆိုတဲ့ type ထဲမှာ "id", "name", "about", "address" ဆိုပြီး Property (င) မပျိုးရှိပါတယ်။

"about", "address" ပဲ User Type ထဲက ထုတ်ပစ်ချင်တယ်ဆိုရင် Omit ကို သုံးပြီး ရေးသားနိုင်ပါတယ်။

Omit<User, "about" | "address"> ရေးသားနည်းနဲ့ UserSummary ဆိုတဲ့ Type အသစ်ကို ဖန်တီးလိုက်တာပဲဖြစ်ပါတယ်။



TS - Partial

```
type NewPartialType = Partial<OriginalType>
```

Partial Utility Type ဆိတာ Object Type တစ်ခုရဲ့ Property အားလုံး Optional Type ဖြစ်အောင်ပြောင်းပေးတာဖြစ်ပါတယ်။

```
type User = {
    id : string;
    name : string;
    about : string;
    address : string;
}
```

```
type PartialUser = {
    id ?: string;
    name ?: string;
    about ?: string;
    address ?: string;
}
```

```
type PartialUser = Partial<User>
```

Partial ရေးသားနည်းနဲ့ PartialUser Type အသစ်ကို ဖန်တီးလိုက်တာပဲဖြစ်ပါတယ်။



TS - Required

```
type RequiredType = Required<OriginalType>
```

Required ကတေသူ Partial နဲ့ ကြောင်းပြန်ပါပဲ။

Object Type ထဲမှာ Optional Property တွေပါနေရင် အားလုံးကို Required ပုံစံကြောင်းပေးပါတယ်။

```
type User = {
    id : string;
    name : string;
    about ?: string;
    address ?: string;
}

type RequiredUser = {
    id : string;
    name : string;
    about : string;
    address : string;
}

type RequierdUser = Required< User >
```

Required ရေးသားနည်းနဲ့ RequiredUser ဆိုတဲ့ Type အသစ်ကို ဖန်တီးလိုက်တာပဲ ဖြစ်ပါတယ်။

TS - Record

```
type RecordType = Record<KeyType, ValueType>
```

string | number | symbol | union

TypeScript မှာ Record ဟာ Object Type တွေကို ဖန်တီးဖို့ အသုံးများတဲ့ Utility Type တစ်ခုပဲ ဖြစ်ပါတယ်။

key-value တွေကို Object ပြုလုပ်တဲ့အခါ Index Signature ကို အသုံးပြုခဲ့တာ လေ့လာ ခဲ့ပြီးဖြစ်ပါတယ်။

```
type RcType = Record< string , string >
```

```
type RcType = {
  [index: string] : string;
}
```

key type ရှိ string နဲ့ value type ကို string ဖြစ် object တစ်ခုကို တည်ဆောက်ချင်ရင်

Record<string, string> ရေးသားနည်းသုံးပြီး ရှင်းလင်းစွာ တည်ဆောက်နိုင်ပါတယ်။

```
type RcType = Record< "id" | "name" , string >
```

```
type RcType = {
  id : string ;
  name : string ;
}
```

Record မှာလည်း အခုလို string union ကို အသုံးပြုပြီး ရေးသားနိုင်ပါတယ်။

TS - Readonly

```
type ReadonlyType = Readonly<OriginalType>
```

Readonly Utility Type က ရှုပြုသား Object Type ရဲ့ Property တွေကို read-only (ဖတ်လိုပဲရအောင်) ပြုလုပ်ပြီး Type အသစ်တစ်ခု ပြန်ပေးပါတယ်။

```
type User = {
    id : string;
    name : string;
    about ?: string;
    address ?: string;
}

type ReadonlyUser = {
    readonly id : string;
    readonly name : string;
    readonly about : string;
    readonly address : string;
}

type ReadonlyUser = Readonly<User>
```

Readonly ရေးသားနည်းနဲ့ ReadonlyUser ဆိုတဲ့ Type အသစ်ကို ဖန်တီးလိုက်တာပဲ ဖြစ်ပါတယ်။



Other Topic

TypeScript Narrowing

Type တွေက Union အနေနဲ့ (၂) မျိုး၊ (၃) မျိုးဖြစ်နေတတ်ပါတယ်။

Code Block အတွင်းမှာ ဘယ် Type ဖြစ်နေပြီလဲဆိတာကို တိတိကျကျသိသွားအောင်လုပ်တာကို Narrowing လုပ်တယ်လို့ ခေါ်ပါတယ်။

Narrowing ကို အဓိကအားဖြင့် Conditional Statements တွေနဲ့ လုပ်နိုင်ပါတယ်။

```
index.ts x
index.ts > [o] convertNumber
1 const convertNumber = (value: string | number) => {
2
3   if (typeof value === "string") { -----}
4
5     // Value type will be "string" in this scope
6
7     console.log(value.toUpperCase());
8
9       (parameter) value: string
10      console.log(value);
11
12
13 } else { -----}
14
15   // Value type will be "number" in this scope
16
17   console.log(value);
18
19   console.log(value.toLocaleString());
20
21 }
22
23 // Value Type will be "number" or "string" in this scope
24 console.log("String or Number: ", value);
25 }
```

value မှာ string အဖြစ်စစ်ထားပြီးရှိလို့ TypeScript သည် String methods တွေကိုသာ Suggest လုပ်ပါတယ်။

else scope ထဲမှာ value မှာ number type ဖြစ်သွားပါပြီ။

`typeof value === "string"` နဲ့ စစ်ထားတဲ့ block အတွင်းကို ကြည့်လိုက်ပါ။ အဲဒီ Condition ကြောင့် value က string type မှန်း TypeScript က သိသွားပြီး narrow down လုပ်လိုက်ပါတယ်။ ဒီတော့ value မှာ string နဲ့ ဆိုင်တဲ့ method တွေပဲ TypeScript Server က feedback ပေးပါတယ့်မယ်။

else block အတွင်းမှာလည်း TypeScript က narrow down လုပ်လိုက်ပါတယ်။ အပေါ်မှာက String Type ဖြစ်တဲ့အတွက် အောက် Block မှာ Number Type အဖြစ် Narrowing လုပ်သွားပါတယ်။



```

index.ts U ×
index.ts > [e] getData
1  type Admin = {
2    role: "admin";
3    adminData: string;
4  };
5
6  type User = {
7    role: "user";
8    userData: string;
9  };
10
11 const getData = (user: Admin | User) => {
12   switch (user.role) {
13     case "admin": {
14       (parameter) user: Admin
15       console.log(user.adminData);
16
17       break;
18     }
19
20     case "user": {
21       console.log(user.userData);
22
23       break;
24     }
25     default:
26       break;
27   }
28 };
29

```

case “admin” တအတွင်းမှာရှိတဲ့
user မှာ Admin Type ထို့ TypeScript မှ
Narrow လုပ်ပေးလိုက်ပါတယ်။

Switch Statement နဲ့ Narrowing လုပ်တဲ့ Code ကိုရေးသားထားခြင်းပဲဖြစ်ပါတယ်။

Admin type အတွက် role မှာ “admin” ဖြစ်ပြီး User type အတွက် role မှာ “user” ဖြစ်ပါတယ်။

“user.role” ကို အခြေခံပြီး switch မှာ စစ်ထွက်ထားတာပါ။

case “admin” block ထဲမှာ user မှာ Admin Type အဖြစ် narrow လုပ်သွားတာကို သိနိုင်ပါတယ်။

ဒီလောက်ဆို Narrowing လုပ်တယ်ဆိုတာကို အခြေခံအားဖြင့် နားလည်သွားလောက်ပါပြီ။



TypeScript Predicate Guard

```
( param : type ) . => param is TargetType ;
```

"is" Type Predicates ကို type narrowing လုပ်ဖို့ အသုံးပြုပါတယ်။

function တစ်ခုက parameter တစ်ခုကို လက်ခံပြီး Narrowing လုပ်ဖို့အတွက် ကိုယ် ပြောင်းလဲချင်တဲ့ Target Type တစ်ခုကို သတ်မှတ်ပေးလိုက်တာဖြစ်ပါတယ်။

ရေးသားနည်း Syntax အရ (param: type) => param is TargetType ; ဖြစ်ပေါ်မယ်

အမှန်တကယ် Return type က Boolean ပဲ ပြန်ရပါမယ်။

```
index.ts U ×
index.ts > [e] actionForAdmin
1  type NormalUser = {
2    role: "user";
3    userInfo: string;
4  };
5
6  type AdminUser = {
7    role: "admin";
8    adminInfo: string;
9  };
10
11 const checkIsAdmin = (user: AdminUser | NormalUser): user is AdminUser => {
12   return user.role === "admin";
13 };
14
15 const actionForAdmin = (user: AdminUser | NormalUser) => {
16   if (checkIsAdmin(user)) [
17     // user is now "admin"
18     (parameter) user: AdminUser
19     console.log(user.adminInfo);
20   ] else {
21   }
22 };
23
24 };
25
```

Return type ↗ Boolean

နူးမှုနာ Code မှာ checkIsAdmin ဆိုတဲ့ function ရှိပါတယ်။

ရည်ရွယ်ချက်က checkIsAdmin ကို ခေါ်လိုက်တာနဲ့ လက်ခံလိုက်တဲ့ user parameter အခြေအနေက true ဖြစ်ခဲ့ရင် သူရဲ့ Type ကို AdminUser type အဖြစ်ပြောင်းပေးလိုက် မှာပါ။

actionForAdmin မှာ လက်ခံထားတဲ့ user parameter က အစမှာ AdminUser (သို့မဟုတ် |) Normal User ပါ။

if (checkIsAdmin(user)) block အတွင်းမှာ user ကို mouse ဖြင့် ထောက်ပြီး ကြည့်ကြည့်တဲ့အပါ AdminUser type အဖြစ် Assertion လုပ်သွားမှာဖြစ်ပါတယ်

TypeScript Template Literal

Template Literal သော string literal types တွေကို ပေါင်းစပ်ပြီး dynamic type တွေ ဖန်တီးနိုင်တဲ့ feature တစ်ခုပဲဖြစ်ပါတယ်။

```
type TargetType = `${string}`
```

Hello, နဲ့စမယ် ကြားထဲ ကြိုက်တယ့်ဘသာ: String ကို အသုံးပြုနိုင်ပြီး နောက်ဆုံးမှာ (!) နှိပ်တယ့် Template Literal Type လေးဖန်တီးကြည့်ပါမယ်။

```
type Greeting = `Hello, ${string}!`  
  
const helloWorld : Greeting = `Hello, World!` ✓ ,," ကော "!" နှစ်မျိုးလုံးပါဝင်ပါတယ်။  
  
const wrongWorld : Greeting = `Hello Wrong` ✗ ,," ကော "!" နှစ်မျိုးလုံးမပါဝင်ပါ။
```



Greeting Template Literal Type လေးဖန်တီးထားပါတယ်။

ထုတေသန helloWorld variable ကတော့ အဆင်ပြပါတယ်။

ဒုတိယ wrongWorld variable ကတော့ “,” “!” တွေ မပါဝင်တဲ့အတွက် Error Feedback ပြန်ရမှာ ဖြစ်ပါတယ်။

Union String Literal တွေ သုံးပြီးတော့လည်း ရေးသားနိုင်ပါတယ်။

```
type UnionName = "Lwin" | "Moe";

type UnionGreeting = `Hello, ${UnionName}!`  

const greet1 : UnionGreeting = `Hello, Lwin!` ✓ "Lwin" နဲ့ "Moe"  

const greet2 : UnionGreeting = `Hello, Moe!` ✓ နဲ့ပဲထည့်သွင်းနိုင်သည်။
```

တကယ့်လက်တွေ့မှာ အသုံးပြုလေ့ရှိတယ့် Code Example fetchAPI ဆိုတဲ့ Function ကိုလေ့လာကြည့်ပါမယ်။

```
ts index.ts 1, U X
ts index.ts > ...
1   type StartWithApi = `/api/${string}`;
2
3   const fetchAPI = (url: StartWithApi) => {};
4
5   fetchAPI("/api/hello");
6
7   fetchAPI("/get-user"); ✘ "/api" နဲ့ စိစိုးလို့အပ်ပါသည်။
8
```



၏ဂုံး

ဒီစာအုပ်လေးက JavaScript အထိက်အလောက်ရပြီး TypeScript ကို စပြီး မထိခဲဖြစ်နေတဲ့သူတွေအတွက် ရေးသားဖြစ်ခဲတာပါ။

တာအုပ်ပါ အကြောင်းအရာ အားလုံးကိုလေ့လာပြီးသွားရင် TypeScript မှာ Beginner ထက် ကျော်သွားပြီလို့ ဆိုရမှာပါ။ သို့ပေါ်မယ်လည်း Conditionals Type Checking, Class Decorators တွေ အစရှိသဖြင့် Intermediate ၁ Advanced Level အထိ ကိုယ့်ဘာသာ ဆက်လက် လေ့လာရမှာတွေ ကျွန်ုပြီးနေပါသေးတယ်။

TypeScript Baby တာအုပ် အကူအညီနဲ့ ဆက်လက်ပြီး ကိုယ့်ရဲ့ Roadmap ကို အကောင်အထည်ဖော်နိုင်ဖို့ ဆုမွန်ကောင်းတောင်းပေးလိုက်ရပါတယ်ခင်ဗျာ။

၂၀၂၅ ခုနှစ်၊ ဖေဖော်ဝါရီ (၁၂) ရက်နေ့တွင် ရေးသားပြီးစီးပါသည်။