

Rapid Application Development

MERN කියන්නේ **JavaScript-based Full Stack Web Development technology stack** එකකි. මෙය **MongoDB, Express.js, React.js, Node.js** කියන technologies 4කින් සමන්විත වේ.

TypeScript

🧠 TypeScript කියන්නේ මොකක්ද?


- TypeScript කියන්නේ Microsoft සමාගමේ නිපැයුණ කළ open-source programming language එකකි.
- මෙය JavaScript එකේ superset එකකි, JavaScript වල ලියනු ලබන code එක TypeScript වල valid වනවා.
- static typing – කියන්නේ variable එකට type එකක් කියලා දැක්විය හැක.

🔑 TypeScript වල විශේෂත්වයන්

විශේෂත්වය

විස්තරය

✅ JavaScript compatible	JavaScript වල නියත සෑම valid code එකකම TypeScript වලත් valid.
📄 Static Typing	Variable එකක type එක (e.g., string, number) පෙරදිගින්ම define කරන්න පුළුවන.
🎯 Function parameter types	Function එකට යවන argument වල type එක set කරන්න පුළුවන.
🏠 Return value types	Function එකේ return වන value එකේ type එක define කරන්න පුළුවන.
📘 Better code readability	Code එක clean & understandable වනවා.

 Better debugging

Compile කරන්නේ type error දැක්වේ – runtime error වන chances අඩුයි.

TypeScript Syntax උදාහරණ

```
// Variable with type
let age: number = 25;

// Function with parameter & return type
function greet(name: string): string {
  return "Hello, " + name;
}
```

TypeScript Key Characteristics

1. Static Typing

- Variable එකකට හෝ function එකකට **type** එක දැක්වීමේ හැකියාව තියනවා.
- මකෙනේ **errors** compile වදේදීම හඳුනාගන්න පුළුවන්.

```
let age: number = 25;
```

```
Function add(a:number,b:number):number{
  return a+b;
}
```

```
//error if called with a string add(2,"kamal");
```

2. OOP Features

- JavaScript වගේම, TypeScript එකේ **Object Oriented Programming (OOP)** features තියනවා:
 - class
 - interface

- inheritance

- මෙතේ **maintain** කරන්න ලේසියි, විශාල **applications** සඳහා සුදුසුයි.

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

3. Transpilation (TypeScript → JavaScript)

- TypeScript එක browser එකට ඉක්මනින් **run** වන්න බැහැ. ඒ නිසා ඒක compile කරන්න වනවා JavaScript එකට.
- ඒක **tsc** කියන command එකෙන් කරන්නේ:

```
tsc script.ts    # මෙතේ script.js එක නිපදවයි
```

- script.ts → TypeScript file එක
- script.js → Transpiled (JavaScript) version එක

4. Compatibility with JavaScript Libraries

- JavaScript වල නියත **libraries** (උදාහරණ: jQuery, React, Axios) කිසිම ගැටළුවක් නැතුව TypeScript එකේ භාවිතා කරන්න පුළුවන්.

```
import axios from 'axios'

axios.get('https://api.example.com');
```

TypeScript භාවිතා කිරීමේ වාසි

- ◆ 1. Static Typing

- compile වෙද්දි error නියතවද කියලා හඳුනා ගන්න පුළුවන්.
- ඒ නිසා bugs අවම වනෙවා, debug කරන වලොව ඉතිරි වනෙවා.

```
Function multiply(a:number,b:number):number{

Return a* b;

}

// multiply(2,"kamal");    compile time errors
```

♦ 2. වැඩිදියුණු කළ හැකි **code** සහ **Documentation** (Improved Code Readability & SelfDocumentation)

- variable එකකට හෝ function එකකට වර්ගය දක්වන නිසා, code එක read karana kenata හොඳින් තේරෙනවා.
- වනෙ developer කෙනෙකුට ද කියවලා අවබෝධය ලබා ගන්න ලේසියි.

```
Function greet(user:{name:string,age:number}){

console.log(`hello,${user.name}`);

}
```

♦ 3. ඉක්මනින් වෙනස් කිරීම (**Refactoring**) සහ නවත්නු කිරීම

- code එක complex වනෙ විට type check එක මඟින් වෙනස්කම් සාමකාමීව කළ හැක.

♦ 4. විශාල ව්‍යාපෘති සඳහා වඩාත් සුදුසුයි

- Interface, Enum, Module වැනි දේවල් තිබුනාම, විශාල code එකක් organize කරන්න ලේසියි.
- කණ්ඩායම් සමඟ වැඩ කිරීමේදී consistency රඳවාගන්න පුළුවන්.

TypeScript Install කිරීම

✓ 1. Node.js Download & Install කරන්න

- <https://nodejs.org> වෙත ගිහිණි Node.js install කරගන්න.
- Terminal එකේ check කරන්න:

```
node --version
```

➡ Node version එක පරීක්ෂා කර install වීලා තියන්නේ.

```
PS C:\Users\ADMIN> node --version
```

```
V22.15.0
```

✅ 2. TypeScript Globally Install කරන්න

```
npm install -g typescript
```

- Install වුණද බලන්න:

```
tsc --version
```

▶ TypeScript Compile & Run කිරීම

1. TypeScript File එකක් ලියන්න (e.g., code.ts):

```
ts
```

```
let message: string = "Hello TypeScript!";
```

```
console.log(message);
```

2. Compile කරන්න:

```
tsc code.ts
```

➡ මෙයින් code.js කියන JavaScript file එකක් නිර්මාණය වේ.

3. Run කරන්න (JS file):

```
node code.js
```

➡ Output: Hello TypeScript!

🧠 Type Annotations කියන්න මොකද්ද?

👉 TypeScript එකේදී, variable එකකට හෝ function එකකට **type** එකක් පැහැදිලිව (explicitly) දෙනවා.

එමඟින්:

- Compile කරන වලොවේම error read karnna පුළුවන්.
- Code එක තවත් කියවන්න ලේසි වෙනවා.
- වැඩි maintainability ලැබෙනවා.

📌 උදාහරණය 1: Variable Type Annotation

ts

```
let name: string = "Lakmal";
```

```
let age: number = 23;
```

```
let isStudent: boolean = true;
```

❌ මගේමේ name වලට string එකක් විතරයි yawann puluwan. Number එකක් assign කළොත් compile error!

📌 උදාහරණය 2: Function Parameter & Return Type Annotation

ts

```
function add(x: number, y: number): number {
```

```
    return x + y;
```

```
}
```

```
console.log(add(5, 3)); // ✅ 8
```

```
console.log(add("5", "3")); // ❌ Compile-time Error
```

- x සහ y වලට number values විතරයි දිය යුතුය.
- Function එක return කරන්නේ **number** එකක් විදියටයි.

Benefits:

- Code එක compile කරදීම වැරදි හඳුනාගන්න පුළුවන්.
- නව developer කනෙක්ටවන් මේ variable/function එකේ අපේක්ෂා වන්නේ මොකක්ද කියලා තේරෙනවා.

♦ any කියන්නේ මොකද්ද?

any type එක variable එකකට assign කරනේ, ඒකට **string, number, boolean, array, object** වගේ **ඕනම data type** එකක් assign කරන්න පුළුවන්.

එකේ type checking එක අඩු වෙනවා. අනිවාර්යයෙන්ම compile-time errors නොමැතිව compile වෙනවා.

උදාහරණයක්:

ts

```
let value: any = "Hello"; // string
```

```
console.log(value);
```

```
value = 100; // number
```

```
console.log(value);
```

```
value = true; // boolean
```

```
console.log(value);
```

```
value = { name: "Lakmal" }; // object
```

```
console.log(value);
```

➡ මගේ `value` කියන variable එකට `string`, `number`, `boolean`, `object` වගේ ඕනම **data type** එකක් **assign** කරනවා.

! නමුත් ගැටලුවක් තියනවා:

`any` type එක ඕනෑම දයෙක් ඇතුලත් කරලා **type safety** එක නැති කරන නිසා, large project එකකට `dangerous` වන්නේ පුළුවන්.

ts

```
let data: any = 123;
```

```
data.toUpperCase(); // ❌ Compile Error නැහැ - Runtime එකේ වැරදි දාලා
```

```
// මයෙ number එකක් වුණත්, toUpperCase() කියන method එක string එකකට ව්තරක් valid!
```

◆ Type Inference කියන්නේ මොකද්ද?

TypeScript එක automatically variable එකක, parameter එකක, නැත්තං return value එකක **data type** එක හඳුනාගන්න කිරමයකි.

➡ ඒ කියන්නේ ඔබට `: string`, `: number` වගේ විශේෂයෙන් type එකක් ලියන්න ඕන නැ, TypeScript එකම first assign කරන value එක බලලා එය guess කරනවා.

◆ Example 1: Variable Type Inference

ts

```
let message = "Hello, Lakmal!";
```


- ♦ මගේ `message` කියන variable එකට `"Hello, Lakmal!"` කියන `string` එකක් assign කරලා තියනේ නිසා
➡ TypeScript එක `message` එක `string` type එකක් කියලා හඳුනාගන්නවා.
-

◆ Example 2: Number Inference

ts

```
let count = 100;
```

- ♦ මගේ `count` එක `number` type එකක් බවට පත්වෙනවා, ඔයාට `: number` කියලා ලියන්න ඕන නෑ.
-

◆ Example 3: Boolean Inference

ts

```
let isLoggedIn = true;
```

- ♦ මෙය TypeScript එකේ `boolean` type එකක් ලෙස හඳුනාගන්නවා.
-

◆ Example 4: Function Return Type Inference

ts

```
function add(x: number, y: number) {  
    return x + y;  
}
```

- ♦ මගේ `return` type එක `number` කියලා TypeScript එක තමාටම හඳුනා ගන්නවා.
-

✓ ලකුණු කරගන්න:

- ඔබ variable එකක් declare කරන විට value එකක් දාලා තිබ්බොත්, TypeScript එකට type එක infer (ගණනය) කරන්න පුළුවන්.
- මේක beginners ලට වඩා පහසුයි.
- නමුත් complex systems වලදී **explicit types** දීම safe.

Array

◆ 1. Array Syntax

✓ Square Brackets Syntax

ts

```
let fruits: string[] = ['Apple', 'Banana'];
```

📌 මගේ `fruits` කියන array එක `string` only values (අක්ෂර) පමණක් භාවිතා කරන්න ඉඩ ඇත.

✓ Generic Type Syntax

ts

```
let ids: Array<number> = [1, 2, 3];
```

📌 මගේ `ids` කියන array එක `number` values පමණක් තැබිය හැක. `Array<type>` කියන generic syntax එක.

◆ 2. Accessing & Modifying Arrays

■ Access an element

ts

```
let fruit = fruits[0]; // "Apple"
```

■ Add an element

ts

```
fruits.push('Grape'); // ['Apple', 'Banana', 'Grape']
```

■ Remove last element

ts

CopyEdit

```
fruits.pop(); // removes 'Grape'
```

◆ Extra Tips

Action	Code
Get length	<pre>fruits.length</pre>
Check includes	<pre>fruits.includes('Apple')</pre>
Loop through items	<pre>for (let f of fruits) {}</pre>

Conditional Flow

TypeScript (and JavaScript) වල **Conditional Flow** කියන්නේ program එකේ logic එක decision-making කිරීමකට පදනම්ව flow වනෙ හැටි - if, else, switch, ternary operator වගේ constructs භාවිතා කිරීමයි.

✓ 1. if...else Statement

ts

```
let age: number = 20;

if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

📌 if condition එක true නම් පළවෙනි block එක, නැත්නම් else block එක run වේ.

✓ 2. else if Statement

ts

```
let score: number = 85;

if (score >= 90) {
    console.log("Grade A");
} else if (score >= 75) {
    console.log("Grade B");
} else {
    console.log("Grade C or lower");
}
```

```
}
```

📌 Multiple conditions check කරන්න `else if` එක use කරන්න සුදුසු.

✅ 3. Ternary Operator

ts

```
let isMember: boolean = true;
```

```
let fee = isMember ? 100 : 200;
```

```
console.log(fee); // Outputs: 100
```

📌 `condition ? valueIfTrue : valueIfFalse` – short form `if...else`

✅ 4. switch Statement

ts

```
let day: number = 3;
```

```
switch (day) {
```

```
  case 1:
```

```
    console.log("Monday");
```

```
    break;
```

```
  case 2:
```

```
    console.log("Tuesday");
```

```
    break;
```

```

case 3:

    console.log("Wednesday");

    break;

default:

    console.log("Another day");

}

```

📌 **switch-case** statement එක එක value එකකට match වුණොම එයට අදාළ block එක execute වෙනවා.

◆ **Type Guards** කියන්නේ මොකක්ද?

TypeScript වලදී object එකක real type එක run time එකේදී check කරලා, program එකේ වරදලා data type එකක් හාවිත නොවෙන්න ඍරක්ෂිත කරමයක් තමයි **type guards**.

◆ **1. typeof Guard – primitive types වලට**

Ts

```

function printValue(value: number | string) {

    if (typeof value === "string") {

        console.log("It's a string: " + value.toUpperCase());

    } else {

        console.log("It's a number: " + value.toFixed(2));

    }

}

printValue("hello");

```

```
printValue(42);
```

● `typeof` use කරලා, `string` ද `number` ද කියලා හඳුනාගෙන එයට ඇරිඳින `function call` කරනවා.

Functions

◆ 1. Basic Function with Parameter and Return Type

ts

```
function add(x: number, y: number): number {  
    return x + y;  
}
```

```
const result = add(5, 3); // 8
```

- ◆ `x` සහ `y` `number` ලෙස declare කරනවා
- ◆ `function` එක `number` return කරනවා

◆ 2. Void Return Type (කිසිම `value` එකක් return නොකරන functions)

Ts

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

- ♦ `void` return type එකේ නම්, return value එකක් නැහැ.
-

◆ 3. Optional Parameter (`?` symbol එකම)

ts

```
function greetUser(name?: string): void {  
    if (name) {  
        console.log("Hello, " + name);  
    } else {  
        console.log("Hello, guest");  
    }  
}
```

- ♦ `name?: string` කියන්නේ optional parameter එකක්.
 - ♦ නැම වටම check කරන්න වෙනවා ඒක තියනෙවද කියලා (`if (name)`)
-

◆ 4. Default Parameter

ts

```
function greetPerson(name: string = "Guest"): void {  
    console.log("Hi " + name);  
}  
  
greetPerson();           // Hi Guest  
  
greetPerson("John");    // Hi John
```

- ♦ `name` parameter එකට default value එකක් `"Guest"` දෙනවා.

! Important Rules

1. Required parameters must come first

(අනිවාර්ය parameter එකකට පස්සින් optional/default ඇතුළත් error)

✓ `function info(name: string, age?: number)`

✗ `function info(age?: number, name: string)`

2. Cannot be both optional and have a default

✗ `function greet(name?: string = "Guest")` - error

✓ Use only one: `name?: string` or `name: string = "Guest"`

◆ 5. Check if Parameter is Provided or Not

ts

```
function logMessage(message?: string) {  
    if (message !== undefined) {  
        console.log("You said: " + message);  
    } else {  
        console.log("No message provided");  
    }  
}
```

- ◆ Optional parameter check කැඳදී `undefined` check එක වඩා safe.
-

✓ Summary:

Concept	Syntax Example
Return Type	<code>function add(x: number, y: number): number</code>
Void Return	<code>function greet(): void</code>
Optional Parameter	<code>function greet(name?: string)</code>
Default Parameter	<code>function greet(name: string = "Guest")</code>
Rules	Required first, no both optional + default

✓ Basic Arrow Function Syntax:

ts

```
const add = (x: number, y: number): number => {  
  return x + y;  
};
```

- ◆ `const add` → variable එකකට function එක assign කරනවා
 - ◆ `(x: number, y: number)` → parameters
 - ◆ `: number` → return type
 - ◆ `=> { return ... }` → arrow function body
-

✓ Shorter Version (if single return):

ts

```
const square = (n: number): number => n * n;
```

- ◆ එක line එකක් return කරන්නම් {} සහ return අවශ්‍ය නැහැ.
-

✓ No Parameters:

ts

```
const greet = (): void => {  
    console.log("Hello!");  
};
```

- ◆ parentheses වලට කිසි parameter එකක් නැහැ
-

✓ Optional & Default Parameters in Arrow Functions:

ts

```
const greetUser = (name: string = "Guest"): void => {  
    console.log("Hi " + name);  
};
```

```
greetUser();           // Hi Guest
```

```
greetUser("Lakmal"); // Hi Lakmal
```

✓ With Optional Parameter:

ts

```
const sayHi = (name?: string): void => {  
  if (name) {  
    console.log("Hi " + name);  
  } else {  
    console.log("Hi anonymous");  
  }  
};
```

◆ Comparison Table:

Function Type	Example
Normal Function	<pre>function add(x: number, y: number): number</pre>
Arrow Function	<pre>const add = (x: number, y: number): number => x + y;</pre>

◆ Interface කියන්නේ මොකක්ද?

Used to define the shape of an object

object එකකට blueprint ekak.

```
interface Person {
```

```
    name: string;


    age: number;
}
```

```
const student: Person = {

    name: "Lakmal",

    age: 23,

};
```

 **Interface** එකක් අනෙක් එකකින් දිගුම (extend) ගන්න:

ts

```
interface Animal {

    name: string;

}
```

```
interface Dog extends Animal {

    breed: string;

}
```

```
const myDog: Dog = {

    name: "Buddy",

    breed: "Labrador",

};
```

◆ Interface සහ Type Alias කියන්නේ මොනවද?

TypeScript එකේ අපිට "type" එකකට නමක් දිලා future එකේ එය නැවත භාවිතා කරන්න පුළුවන්. ඒ සඳහා **Interface** එකක් හෝ **Type Alias** එකක් භාවිතා කරනවා.

⚙️ Interface vs Type Alias

Feature	Interface	Type Alias
Object Types	✅ (එකට main use එක නමයි)	✅ (object type එකක් represent කරන්න පුළුවන්)
Primitives (number, string වගේ)	❌ (support නැහැ)	✅ (primitive type එකකට නමක් දිය හැක)
Union / Intersection Types (එකට වඩා types combine කරන එක)	❌ (union නැ), ✅ (intersection එක extends හරහා)	✅ union
Extends / Implements (type එකක් inherit කරන එක)	✅ (interface එක extend කරන්න පුළුවන්)	✅ (extend කරන්නේ & හරහා)

🧠 උදාහරණ:

1. Object Type එකක්

```
ts
```

```
// Interface
```

```
interface Person {  
    name: string;  
    age: number;  
}
```

```
// Type Alias
```

```
type PersonType = {  
    name: string;  
    age: number;  
}
```

2. Primitive Type එකක් (Type Alias වලට විතරයි හැකියාව තියන්නේ)

```
ts
```

```
type ID = string;
```

3. Union Type (Type Alias වලට විතරයි)

```
ts
```

```
type Status = "success" | "error" | "loading";
```

4. Extend කරන එක (දකුණේම පුළුවන්)

```
ts
```

```
// Interface extend
```

```
interface Animal {  
    name: string;
```

```

}

interface Dog extends Animal {

    breed: string;

}

// Type Alias extend (intersection)

type Animal = {

    name: string;

}

type Dog = Animal & {

    breed: string;

}

```

Define Objects Using Interfaces

✅ Interface කියන්නේ මොකක්ද?

TypeScript එකේ interface කියන්නේ object එකක:

- required properties,
- type (string, number, boolean, etc.)
- (optional) properties සහ methods

දැක්වීමට භාවිතා කරන structure එකකි.

♦ Interface එකක හැමදෙයෙකිම define කිරීමෙන් ලැබෙන වාසි:

- Code එක clear & structured වේ.

- Object එක type-safe වේ (වැරදි data format එකක් use කරනොත් error එනවා).
- Auto-complete සහ developer-friendly coding environment එකක් ලැබෙනවා.

◆ 1. Basic Interface (මූලික Interface එකක්)

ts

```
interface Person {  
    name: string;  
    age: number;  
}
```

මෙහිදී **Person** කියන interface එක object එකකට **name** (string) සහ **age** (number) කියන properties තියනේන කියලා කියනවා.

◆ 2. Creating an Object with the Interface

ts

```
const user: Person = {  
    name: "Lakmal",  
    age: 23  
};
```

user කියන object එක **Person** interface එකට අනුකූලව තියනේන ඕන.

◆ 3. Optional Property (අමතර property එකක්)

ts

```
interface Person {
```

```
    name: string;

    age: number;

    email?: string; // optional property
}
```

ts

```
const user1: Person = {

    name: "Lakmal",

    age: 23

};
```

```
const user2: Person = {

    name: "Kamal",

    age: 30,

    email: "kamal@example.com"

};
```

◆ 4. Method in Interface (Function එකක් interface එකට දාන්න)

ts

```
interface Person {

    name: string;

    age: number;

    greet(): void; // method

}
```

```
ts

const user: Person = {

  name: "Lakmal",

  age: 23,

  greet() {

    console.log("Hello, I'm " + this.name);

  }

};

user.greet(); // Hello, I'm Lakmal
```

◆ 5. Nested Interfaces (අනුලේඛ object එකක් interface එකක ඇතුළත)

```
ts

interface Address {

  city: string;

  country: string;

}

interface Person {

  name: string;

  age: number;

  address: Address; // nested interface
```

```
}
```

ts

```
const user: Person = {  
  name: "Lakmal",  
  age: 23,  
  address: {  
    city: "Monaragala",  
    country: "Sri Lanka"  
  }  
};
```

◆ 6. Extending Interface (Interface එකක් inherit කිරීම)

ts

```
interface Person {  
  name: string;  
  age: number;  
}  
  
interface Employee extends Person {  
  employeeId: string;  
}
```

ts

```
const emp: Employee = {  
  name: "Lakmal",  
  age: 23,  
  employeeId: "EMP001"  
};
```

Employee කියන interface එක Person එක expand කරනවා.

7. Multiple Interface Extension (Interfaces කිහිපයක් extend කිරීම)

ts

```
interface Person {  
  name: string;  
}
```

```
interface Worker {  
  jobTitle: string;  
}
```

```
interface Engineer extends Person, Worker {  
  skills: string[];  
}
```

ts

```
const eng: Engineer = {
  name: "Lakmal",
  jobTitle: "Software Engineer",
  skills: ["Java", "TypeScript"]
};
```

Summary (කෙටි සාරාංශය):

Concept	Description
<code>interface</code>	Object එකක් define කරන්න පාවිච්ඡා කරනවා
Required properties	අවශ්‍ය fields (no <code>?</code>)
Optional properties	<code>?</code> symbol එකේ දැක්වෙනවා
Method	Function එකක් interface එකේ declare කරනවා
Nested interface	Interface එකක් තුළ වෙනත් interface
<code>extends</code>	Interface එකක් තවත් එකක් inherit කරනවා
Multi-extends	Interfaces කිහිපයක් extend කරනවා

What is an Enum? (Enum කියන්නේ මොකක්ද?)

Enum = "Enumeration" → එක group එකක් විදියට **related constant values** එකට තියලා **easy-to-read names** assign කරන ක්රමයක්.

ts

```
enum Direction {  
  
    North,  
  
    South,  
  
    East,  
  
    West  
}
```

Default වලට North = 0, South = 1, East = 2, West = 3 වගේ **numeric values** assign වෙනවා.

◆ 1. Numeric Enum (අංක ලෙස enum)

ts

```
enum Status {  
  
    Pending,    // 0  
  
    Approved,   // 1  
  
    Rejected    // 2  
}
```

Ts

```
let s: Status = Status.Approved;  
  
console.log(s); // Output: 1
```

◆ 2. Setting a Starting Value (ආරම්භ value එක **set** කිරීම)

ts

```
enum Status {  
    Pending = 5,    // start from 5  
    Approved,       // 6  
    Rejected        // 7  
}
```

TypeScript auto-increments the next ones.

◆ 3. Assigning Explicit Values (සෑම එකකටම **manually value assign** කිරීම)

ts

```
enum Status {  
    Pending = 10,  
    Approved = 20,  
    Rejected = 99  
}
```

මෙතෙක් auto-increment එකක් නැහැ. ඔබටම values define කරන්න වෙනවා.

◆ 4. String Enums (Starting Enums with strings)

String enums වලදී value එකක් assign කරන්න අනිවාර්යයි.

ts

```
enum Direction {  
    North = "N",  
    South = "S",  
    East = "E",  
    West = "W"  
}
```

ts

```
let d: Direction = Direction.North;  
console.log(d); // Output: "N"
```



Summary Table

Feature	Example	Output
Numeric enum	<code>enum A { X, Y }</code>	<code>X = 0, Y = 1</code>
Set starting value	<code>enum A { X = 5, Y }</code>	<code>X = 5, Y = 6</code>

Explicit values

```
enum A { X = 10, Y = 20 }
```

X = 10

String enum

```
enum A { X = "One", Y = "Two" }
```

X = "One"

Tuples කියන්නේ මොකක්ද?

Tuple එක් කියන්නේ:

- **fixed-length**
- **ordered**
- **heterogeneous** (වෙනස් වර්ගවල values තියන array එක්).

ts

```
let person: [string, number] = ["Lakmal", 23];
```

මගේ **string** value එක මූලික, **number** value එක දවෙනි.

Tuples vs Arrays

Feature	Tuple	Array
Length	Fixed	Variable
Types	Heterogeneous (mixed)	Usually Homogeneous (same)

Order
enforced

✔ Yes

✗ No

◆ 1. Basic Tuple Example

ts

```
let user: [string, number] = ["Kamal", 30];
```

මෙම `user` variable එකට පළවන value එක `string`, දෙවන value එක `number` විය යුතුය.

◆ 2. Rest Element in Tuple (බාගත Tuple එකකට `rest` භාවිතය)

ts

```
let values: [number, ...string[]] = [1, "two", "three", "four"];
```

මගේ:

- පළවන element එක `number`
 - ඉතිරි හැම එකම `string` (rest element)
-

◆ 3. Labeled Tuples (Tuple එකට නාම දැමීම)

Labels කියන්නේ developer-friendly naming එකකි. ඒක TypeScript වලදී Tooltip එකේ දී helpful වෙයි.

ts

```
type Point = [x: number, y: number];
```

```
const p: Point = [10, 20];
```

මගේ x සහ y කියන labels කියන්නේ documentation/tooltip වලට පමණයි - compile වනකොට disappear වෙනවා.

◆ 4. Optional Tuple Elements (Optional elements with ?)

ts


```
type Contact = [name: string, email?: string];  
  
let c1: Contact = ["Lakmal"];  
  
let c2: Contact = ["Lakmal", "lakmal@example.com"];
```

◆ Generics කියන්නේ මොකක්ද?

Generics කියන්නේ:

- **Reusable**: එක function/class එකක් බොහෝ types වලට යළි යොදා ගන්න පුළුවන්
- **Flexible** : වෙන වෙනම data shapes වලට support
- **Type-Safe** : compile time එකේදීම type errors පරීක්ෂා කරනවා

✓ Benefit Table

Benefit	Description
 Reusability	එක වර ලියලා, බොහෝ types වලට යොදාගන්න පුළුවන්

✔ Type Safety

Compile-time එකතුවේදී error check කරලා අවදානම අඩු කරනවා

🔧 Flexibility

වෙනස් data shapes වලට adapt වෙනවා

◆ 1. Generic Function

ts

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

```
let num = identity<number>(100);    // returns number
```

```
let str = identity<string>("Lakmal"); // returns string
```

<T> ඔස්සේ generic type variable එක. ඔබට T වනුවට වනෙම නම් දන්නේ පුළුවන් (ex: <U>, <X>).

◆ 2. Generic Class

ts

```
class Box<T> {  
    content: T;  
  
    constructor(value: T) {  
        this.content = value;  
    }  
}
```

```
    getContent(): T {  
        return this.content;  
    }  
}  
  
const stringBox = new Box<string>("Hello");  
console.log(stringBox.getContent()); // "Hello"  
  
const numberBox = new Box<number>(123);  
console.log(numberBox.getContent()); // 123
```

◆ 3. Generic Interface

```
ts  
  
interface KeyValue<K, V> {  
    key: K;  
    value: V;  
}  
  
const kv1: KeyValue<string, number> = {  
    key: "age",  
    value: 23  
};  
  
const kv2: KeyValue<number, boolean> = {
```

```
key: 1,  
value: true  
};
```

Interface එකේදී `<K, V>` වගේ multiple generic types ගාවිතා කරන්න පුළුවන්.

◆ 4. Generic with Constraints (type borders දානවා)

Sometimes, ඔබට type එකක් **limit** කරන්න ඕනේ. එවිට `extends` keyword එක ගාවිතා කරනවා.

ts

```
function printLength<T extends { length: number }>(item: T): void {  
    console.log(item.length);  
}
```

```
printLength("Hello");    // ✔ valid
```

```
printLength([1, 2, 3]);  // ✔ valid
```

```
// printLength(123);     ✖ error: number doesn't have length
```



Summary Table

Concept	Syntax Example
Generic Function	<code>function name<T>(arg: T): T</code>

Generic Class `class Name<T> { ... }`

Generic Interface `interface X<K, V> { ... }`

Constraint `<T extends Type>`

♦ What is a Module? / මොකද්ද Module එකක් කියන්නේ?

- Any file with a top-level **import** or **export** is a module.
👉 **import** ගේ export එකක් නියමයෙන් එකකින් කියන්නේ module එකක්.
- Module එකකට තමන්ගේම **scope** එකක් තියනවා.
👉 ඒ කියන්නේ අනෙක් file වලට effect වන global pollution එකක් නැහැ.
- Module භාවිතා කරන ප්රධාන හේතු:
 - ✅ Code එක හොඳට **organize** කරන්න
 - ✅ Code එක **encapsulate** (ඇතුළත පමණක් access කරන විදිහට) කරන්න
 - ✅ නැවත නැවත **reuse** කරන්න

📦 Export එකක් කරන ආකාරය

- **Named Exports:** එක module එකකින් එකකට වැඩි item export කරන්න පුළුවන්.
- **Default Export:** එක module එකකට එකම **default export** එකක් තියිය යුතුයි.

📦 Import කරන්නේ කොහොමද? / Import Types and Examples

English Name

✅ Syntax

📌 විස්තරය

Named Import	<code>import { add, PI } from './math'</code>	අවශ්‍ය වශයෙන්ම export කිහිපයක් import කරන විදින
Alias Import	<code>import { add as sum } from './math'</code>	Import කරන function එකකට වෙනම නමක් දැන්න
Namespace Import	<code>import * as MathUtils from './math'</code>	සියලුම එක object එකක් විදිනට import කරන්න
Default Import	<code>import multiply from './math'</code>	Module එකේ default export එක import කිරීම (no {})
Type-Only Import	<code>import type { Cat } from './animal'</code>	Runtime එකට effect නොවන types පමණක් import කිරීම

උදාහරණයක් (Example):

math.ts

ts

```
export const PI = 3.14;

export function add(x: number, y: number): number {
    return x + y;
}

export default function multiply(x: number, y: number): number {
    return x * y;
}
```

main.ts

Ts

```
import multiply, { add, PI } from './math';
```

```
console.log(add(2, 3));      // ➡ 5
```

```
console.log(PI);            // ➡ 3.14
```

```
console.log(multiply(2, 3)); // ➡ 6
```



- `import { something } =>` Named import
- `import something =>` Default import
- `import * as name =>` Everything as an object
- `import type { Something } =>` Only type එකක් import කරන එක

