TypeScript Enums

> **Overview** Enums are one of TypeScript's most distinctive features — a way to define a set of named constants that represent a fixed collection of related values. They make your code more readable, self-documenting, and safer by replacing magic strings and numbers with meaningful names. This article covers what enums are, why they exist, the different kinds, when to use them (and when not to), and practical examples throughout.

Table of Contents

- What Is an Enum?
- Why Use Enums?
- Types of Enums
  - Numeric Enums
  - String Enums
  - Heterogeneous Enums
  - Const Enums
- How Enums Work Under the Hood
- Common Patterns and Use Cases
- When NOT to Use Enums
- Enums vs Alternatives
- Summary

What Is an Enum?

An **enum** (short for *enumeration*) is a special TypeScript construct that lets you define a type made up of a **fixed set of named values**.

Think of it as giving friendly names to a group of related constants — instead of scattering `1`, `2`, `3` or `"admin"`, `"user"`, `"guest"` throughout your code, you define them once in one place and refer to them by name.

Basic Syntax

```ts
enum Direction {
  Up,
  Down,
  Left,
  Right
}

const move: Direction = Direction.Up;
```

That's it at its simplest. You define the enum, then use its members like `Direction.Up` anywhere you'd use a value.

Why Use Enums?

Before enums, developers used plain constants or raw strings/numbers. Here's the problem with that:

```
// ❌ Without enums — magic numbers everywhere
function setStatus(status: number) {
  if (status === 1) { /* active */ }
  if (status === 2) { /* inactive */ }
  if (status === 3) { /* pending */ }
}

setStatus(4); // No error — but 4 means nothing!
```

```
// ❌ Without enums — magic strings
function setRole(role: string) {
  if (role === 'admin') { /* ... */ }
}

setRole('admni'); // Typo — no error caught at compile time
```

Enums fix all of this:

```
// ✅ With enums — clear, safe, self-documenting
enum Status {
  Active,
  Inactive,
  Pending
}

function setStatus(status: Status) {
  if (status === Status.Active) { /* active */ }
}

setStatus(Status.Active);  // ✅ Clear and type-safe
setStatus(4);              // ❌ TypeScript error: 4 is not assignable to type 'Status'
```

The core benefits:

Benefit	Description
Readability	`Direction.Up` is clearer than `0` or `"up"`
Type safety	TypeScript prevents invalid values from being passed

Benefit	Description
Autocomplete	IDEs suggest all valid members
Refactoring	Change a value in one place — updates everywhere
Self-documentation	The enum itself describes the full set of valid options

---

Types of Enums

TypeScript has several flavours of enums. Understanding each one is key to using them correctly.

Numeric Enums

The **default** enum type. Members are automatically assigned incrementing numbers starting from `0`.

```
enum Direction {
  Up,    // 0
  Down,  // 1
  Left,  // 2
  Right  // 3
}

console.log(Direction.Up);    // 0
console.log(Direction.Down);  // 1
```

Custom Starting Value

You can override the starting number, and subsequent members will continue from there:

```
enum StatusCode {
  OK = 200,
  Created,       // 201 (auto-increments)
  Accepted,      // 202
  BadRequest = 400,
  Unauthorized,  // 401
  Forbidden,     // 402 ← careful, not 403!
  NotFound = 404
}

console.log(StatusCode.OK);           // 200
console.log(StatusCode.Unauthorized); // 401
console.log(StatusCode.NotFound);     // 404
```

⚠ **Auto-Increment Pitfall When you mix custom values and auto-incremented values, the auto-increment resumes from the last explicitly set value. This can produce unexpected results — always set all values explicitly if using custom numbers.**

Reverse Mapping

Numeric enums have a unique feature: **reverse mapping.** You can look up a name by its value.

```
enum Direction {
  Up,    // 0
  Down,  // 1
  Left,  // 2
  Right  // 3
}

console.log(Direction[0]);          // "Up"
console.log(Direction[Direction.Up]); // "Up"

// Useful for debugging or logging
function logDirection(d: Direction) {
  console.log(`Moving: ${Direction[d]}`);
}

logDirection(Direction.Left); // "Moving: Left"
```

String Enums

Each member is explicitly assigned a **string value.** There is no auto-increment — every member must be initialized.

```
enum Role {
  Admin = 'ADMIN',
  Editor = 'EDITOR',
  Viewer = 'VIEWER'
}

console.log(Role.Admin);  // "ADMIN"
console.log(Role.Viewer); // "VIEWER"
```

Why String Enums Are Often Preferred

```
// Numeric enum — value in logs/debugger tells you nothing
enum Status {
  Active,   // 0
  Inactive  // 1
}
console.log(Status.Active); // 0 ← what does 0 mean?

// String enum — value is self-describing
enum Status {
  Active = 'ACTIVE',
  Inactive = 'INACTIVE'
}
console.log(Status.Active); // "ACTIVE" ← immediately clear
```

String Enums for APIs and Storage When values are serialized to JSON, sent over HTTP, or stored in a database, string enums produce human-readable output. Numeric enums produce raw numbers that lose meaning outside the codebase.

String Enums in Practice

```
enum HttpMethod {
  Get = 'GET',
  Post = 'POST',
  Put = 'PUT',
  Patch = 'PATCH',
  Delete = 'DELETE'
}

function makeRequest(url: string, method: HttpMethod): void {
  console.log(`${method} ${url}`);
  // Output: "GET https://api.example.com/users"
}

makeRequest('https://api.example.com/users', HttpMethod.Get);
```

Heterogeneous Enums

An enum where members have **mixed numeric and string values.**

```
enum Mixed {
  No = 0,
  Yes = 'YES'
}
```

⚠ **Avoid Heterogeneous Enums TypeScript allows them, but they serve almost no practical purpose and make your code confusing. Avoid them unless you have a very specific reason. Stick to pure numeric or pure string enums.**

Const Enums

Declared with the `const` keyword. They are **completely erased at compile time** and inlined as their literal values — resulting in zero runtime overhead.

```
const enum Direction {
  Up,
  Down,
  Left,
  Right
}

const move = Direction.Up;
```

Compiled JavaScript output:

JS

```
// The entire enum disappears — the value is inlined directly
const move = 0; // Direction.Up → 0
```

Compare with a regular enum's compiled output:

JS

```
// Regular enum creates a full runtime object
var Direction;
(function (Direction) {
  Direction[Direction["Up"] = 0] = "Up";
  Direction[Direction["Down"] = 1] = "Down";
  // ...
})(Direction || (Direction = {}));
```

🔥 **When to Use `const enum` Use `const enum` when you need the type-safety benefits of enums but want zero runtime cost — common in performance-sensitive or library code. The trade-off is that you cannot iterate over const enum members or use reverse mapping.**

⚠ **Const Enums and Isolating Declarations If you use `isolatedModules: true` in your `tsconfig` (required by tools like Babel, Vite, or `ts-transpile-module`), `const enum` across module boundaries**

can cause errors. In those cases, use regular enums or union types instead.

How Enums Work Under the Hood

Understanding what TypeScript compiles enums into helps avoid surprises.

Numeric Enum Compilation

```
// TypeScript
enum Color {
  Red,
  Green,
  Blue
}
```

JS

```
// Compiled JavaScript
var Color;
(function (Color) {
  Color[Color["Red"] = 0] = "Red";
  Color[Color["Green"] = 1] = "Green";
  Color[Color["Blue"] = 2] = "Blue";
})(Color || (Color = {}));
```

This creates a **bidirectional object**:

JS

```
Color = {
  0: "Red",
  1: "Green",
  2: "Blue",
  Red: 0,
  Green: 1,
  Blue: 2
}
```

This is why reverse mapping works for numeric enums — the object stores both directions.

String Enum Compilation

```
// TypeScript
enum Status {
  Active = 'ACTIVE',
  Inactive = 'INACTIVE'
}
```

```JS
// Compiled JavaScript
var Status;
(function (Status) {
  Status["Active"] = "ACTIVE";
  Status["Inactive"] = "INACTIVE";
})(Status || (Status = {}));
```

This creates a **one-directional object** (no reverse mapping):

```JS
Status = {
  Active: "ACTIVE",
  Inactive: "INACTIVE"
}
```

String enums **do not** get reverse mapping because multiple members could theoretically share the same string value, making a reverse lookup ambiguous.

Common Patterns and Use Cases

1. Application State / Status

Enums are ideal when a value can only ever be one of a fixed set of states.

```
enum LoadingState {
  Idle = 'IDLE',
  Loading = 'LOADING',
  Success = 'SUCCESS',
  Error = 'ERROR'
}

interface FetchState<T> {
  status: LoadingState;
  data: T | null;
  error: string | null;
}

function handleState<T>(state: FetchState<T>): string {
  switch (state.status) {
    case LoadingState.Idle:    return 'Waiting to fetch...';
    case LoadingState.Loading: return 'Fetching data...';
    case LoadingState.Success: return `Got data: ${JSON.stringify(state.data)}`;
    case LoadingState.Error:   return `Error: ${state.error}`;
  }
}
```

2. User Roles and Permissions

```
enum UserRole {
  SuperAdmin = 'SUPER_ADMIN',
  Admin = 'ADMIN',
  Editor = 'EDITOR',
  Viewer = 'VIEWER',
  Guest = 'GUEST'
}

function canEdit(role: UserRole): boolean {
  return role === UserRole.SuperAdmin ||
         role === UserRole.Admin ||
         role === UserRole.Editor;
}

function canDelete(role: UserRole): boolean {
  return role === UserRole.SuperAdmin || role === UserRole.Admin;
}

const userRole = UserRole.Editor;
console.log(canEdit(userRole));   // true
console.log(canDelete(userRole)); // false
```

3. Switch Statements with Exhaustive Checks

One of the most powerful patterns — TypeScript can verify you've handled **every possible enum value.**

```ts
enum Shape {
  Circle = 'CIRCLE',
  Square = 'SQUARE',
  Triangle = 'TRIANGLE'
}

function getArea(shape: Shape, size: number): number {
  switch (shape) {
    case Shape.Circle:
      return Math.PI * size * size;
    case Shape.Square:
      return size * size;
    case Shape.Triangle:
      return (Math.sqrt(3) / 4) * size * size;
    default:
      // This line will cause a TypeScript error if a new Shape
      // is added but not handled in the switch - exhaustive check!
      const _exhaustive: never = shape;
      throw new Error(`Unhandled shape: ${_exhaustive}`);
  }
}

getArea(Shape.Circle, 5); // ✅
```

🔥 **The `never` Exhaustiveness Pattern Assigning the unhandled value to a `never` type in the `default` branch is a compile-time safety net. If you add `Shape.Pentagon` later and forget to add a `case` for it, TypeScript will throw a type error immediately.**

4. HTTP Status Codes

```ts
enum HttpStatus {
  OK = 200,
  Created = 201,
  NoContent = 204,
  BadRequest = 400,
  Unauthorized = 401,
  Forbidden = 403,
  NotFound = 404,
  InternalServerError = 500
}

function handleResponse(status: HttpStatus): string {
  switch (status) {
    case HttpStatus.OK:
    case HttpStatus.Created:
      return 'Request successful';
    case HttpStatus.BadRequest:
      return 'Invalid request data';
    case HttpStatus.Unauthorized:
      return 'Please log in';
    case HttpStatus.NotFound:
      return 'Resource not found';
    case HttpStatus.InternalServerError:
      return 'Server error - try again later';
    default:
      return 'Unknown status';
  }
}
```

5. Iterating Over Enum Members

```ts
enum Color {
  Red = 'RED',
  Green = 'GREEN',
  Blue = 'BLUE'
}

// Get all keys
const colorKeys = Object.keys(Color);
console.log(colorKeys); // ["Red", "Green", "Blue"]

// Get all values
const colorValues = Object.values(Color);
console.log(colorValues); // ["RED", "GREEN", "BLUE"]

// Build a select/dropdown list
const colorOptions = Object.entries(Color).map(([label, value]) => ({
  label,  // "Red", "Green", "Blue"
  value   // "RED", "GREEN", "BLUE"
}));
```

✎ **Iterating Numeric Enums** Iterating over numeric enums with `Object.keys()` or `Object.values()` will include both the names and the reverse-mapped numbers, which is often not what you want. String enums iterate cleanly.

6. Enums as Function Parameters

```ts
enum SortOrder {
  Ascending = 'ASC',
  Descending = 'DESC'
}

enum SortField {
  Name = 'name',
  CreatedAt = 'createdAt',
  UpdatedAt = 'updatedAt'
}

function sortUsers(field: SortField, order: SortOrder) {
  console.log(`Sorting by ${field} ${order}`);
  // Output: "Sorting by name ASC"
}

sortUsers(SortField.Name, SortOrder.Ascending);    // ✅
sortUsers('name', 'ASC');                          // ❌ TypeScript error
```

7. Enums with Interfaces

```
enum NotificationType {
  Info = 'INFO',
  Success = 'SUCCESS',
  Warning = 'WARNING',
  Error = 'ERROR'
}

interface Notification {
  id: number;
  type: NotificationType;
  message: string;
  timestamp: Date;
}

const notification: Notification = {
  id: 1,
  type: NotificationType.Success,
  message: 'File uploaded successfully',
  timestamp: new Date()
};
```

When NOT to Use Enums

Enums are not always the right tool. Here are situations to reconsider:

1. When the Set Is Open-Ended

If new values can be added dynamically (e.g., user-generated tags, API-driven categories), enums are the wrong choice — they're for fixed, compile-time-known sets.

```
// ❌ Bad - tags can be anything, they're not a fixed set
enum Tag {
  JavaScript,
  TypeScript,
  Angular
  // ... can't enumerate all possible tags
}

// ✅ Better - just use string
type Tag = string;
```

2. When You Need True Constant Objects

If you need an object with rich metadata per entry, a plain `const` object often works better.

```ts
// ❌ Enums can't store metadata
enum Planet {
  Mercury,
  Venus,
  Earth
}

// ✅ A const object holds richer data
const PLANETS = {
  Mercury: { distanceFromSun: 0.39, moons: 0 },
  Venus:   { distanceFromSun: 0.72, moons: 0 },
  Earth:   { distanceFromSun: 1.00, moons: 1 }
} as const;

type Planet = keyof typeof PLANETS;
```

3. When Using Isolate Modules (Babel/Vite)

As mentioned in the `const enum` section, if your build tool doesn't use the TypeScript compiler directly, some enum patterns may not work as expected. Union types are safer in those environments.

Enums vs Alternatives

It's worth knowing what you're choosing between.

Enum vs Union Type

```ts
// Union type approach
type Direction = 'Up' | 'Down' | 'Left' | 'Right';

function move(direction: Direction) { /* ... */ }
move('Up');   // ✅
move('Diag'); // ❌ Error
```

```ts
// Enum approach
enum Direction {
  Up = 'Up',
  Down = 'Down',
  Left = 'Left',
  Right = 'Right'
}

function move(direction: Direction) { /* ... */ }
move(Direction.Up); // ✅
move('Up');         // ❌ Error - must use enum member
```

	Union Type	Enum
Syntax	Lighter	More verbose
Runtime object	No (erased)	Yes
Autocomplete	✅	✅
Iterable	❌	✅
Reverse mapping	❌	✅ (numeric only)
Serialization	Natural strings	Depends on type
Extensibility	Easy to extend	Needs redeclaring

Rule of Thumb

- Use **union types** for simple, lightweight string discriminators — especially in function parameters and interfaces.
- Use **enums** when you need iteration, reverse mapping, a runtime object, or a named set that's shared across many files.

Enum vs `as const` Object

```ts
// as const approach
const Direction = {
  Up: 'Up',
  Down: 'Down',
  Left: 'Left',
  Right: 'Right'
} as const;

type Direction = typeof Direction[keyof typeof Direction];

function move(direction: Direction) { /* ... */ }
move(Direction.Up); // ✅
move('Up');         // ✅ - also works (unlike enums)
```

The `as const` pattern gives you an iterable runtime object and union type safety, but is more verbose to set up and allows raw string literals where enums don't.

Summary

Key Takeaways

What: An enum is a named set of related constants — either numeric or string values — defined in one place and used throughout your code.

Why: Enums prevent magic numbers and magic strings, enforce type safety, enable autocomplete, and make your code self-documenting.

How:

- `enum Foo { A, B, C }` for numeric (auto-incremented)
- `enum Foo { A = 'A', B = 'B' }` for string (explicit values, preferred for serialization)
- `const enum Foo { ... }` for zero-overhead inlining

When to use:

- Fixed sets of related values that won't grow dynamically
- State machines and status flags
- Role/permission systems
- Switch statements where exhaustive checking matters
- When you need to iterate over all valid values

When to avoid:

- Open-ended sets (use `string` or `string[]` instead)
- Environments using Babel/Vite with `isolatedModules` (prefer union types)
- Simple one-off discriminators in a single file (a union type is leaner)

Quick Reference Card

```ts
// Numeric enum (auto-incremented from 0)
enum Direction { Up, Down, Left, Right }
Direction.Up     // 0
Direction[0]     // "Up" (reverse mapping)

// Numeric enum (custom values)
enum Code { OK = 200, NotFound = 404 }

// String enum (explicit values — preferred)
enum Status { Active = 'ACTIVE', Inactive = 'INACTIVE' }
Status.Active    // "ACTIVE"

// Const enum (inlined at compile time, no runtime object)
const enum Flag { On, Off }

// Using an enum as a type
function process(status: Status): void { ... }

// Exhaustive switch
switch (value) {
  case MyEnum.A: ...
  case MyEnum.B: ...
  default:
    const check: never = value; // compile error if a case is missed
}

// Iterating (string enums are clean)
Object.keys(Status)   // ["Active", "Inactive"]
Object.values(Status) // ["ACTIVE", "INACTIVE"]
```