# The TypeScript Handbook

## Enums

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript.

Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

## Numeric enums

We’ll first start off with numeric enums, which are probably more familiar if you’re coming from other languages. An enum can be defined using the enum keyword.

enum Direction {

Up = 1,

Down,

Left,

Right,

}[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAIglgJ2AYwC5wPYigbwFBRQCqADlALxQCMANAbBgO4h2EAywAZqq1AEpwA5gAseeAL5A)

Above, we have a numeric enum where Up is initialized with 1. All of the following members are auto-incremented from that point on. In other words, Direction.Up has the value 1, Down has 2, Left has 3, and Right has 4.

If we wanted, we could leave off the initializers entirely:

enum Direction {

Up,

Down,

Left,

Right,

}[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAIglgJ2AYwC5wPYigbwFBRQCqADgDQGwYDuIFhAMsAGar1QBKcA5gBZt4AvkA)

Here, Up would have the value 0, Down would have 1, etc. This auto-incrementing behavior is useful for cases where we might not care about the member values themselves, but do care that each value is distinct from other values in the same enum.

Using an enum is simple: just access any member as a property off of the enum itself, and declare types using the name of the enum:

enum UserResponse {

No = 0,

Yes = 1,

}

function respond(recipient: string, message: UserResponse): void {

// ...

}

respond("Princess Caroline", UserResponse.Yes);[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAqgzsATgJWHADgexAqBvAKCigDlMoBeKABgBoioBNNSqARnoF8CCAzMEAGMALgEtsURGiwgAJgAopg0elGhhALihxhiUSADmtKBDRwAhgeBb4SVBmwIAlFoBumUbPwMA9D6gAdEEE3ARSDnLyAEQACnpCZlAAwuaImAA2+sBRxrYo0o7AAcxwTgDcQA)

Numeric enums can be mixed in [computed and constant members (see below)](https://www.typescriptlang.org/docs/handbook/enums.html#computed-and-constant-members). The short story is, enums without initializers either need to be first, or have to come after numeric enums initialized with numeric constants or other constant enum members. In other words, the following isn’t allowed:

enum E {

A = getSomeValue(),

B,

Enum member must have initializer.Enum member must have initializer.}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygIwAYBsaBQBjOAO0QBdQBzSUgZTgFtIA1AQwBsBXSUAXlAAoAlLwB8oAEwBmANy4QoALRL8HUkoW5IRDvVABRUAG9coUAEFelanUatOkIQBoToAELOAvkA)

## String enums

String enums are a similar concept, but have some subtle [runtime differences](https://www.typescriptlang.org/docs/handbook/enums.html#enums-at-runtime) as documented below. In a string enum, each member has to be constant-initialized with a string literal, or with another string enum member.

enum Direction {

Up = "UP",

Down = "DOWN",

Left = "LEFT",

Right = "RIGHT",

}[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAIglgJ2AYwC5wPYigbwFBRQCqADlALxQBERAClQDQGwYDu2lVMA8gOoByjZgBlgAM1QVqwgKIAxACpDCAJTgBzABaTOKgJIBxABJKmAXyA)

While string enums don’t have auto-incrementing behavior, string enums have the benefit that they “serialize” well. In other words, if you were debugging and had to read the runtime value of a numeric enum, the value is often opaque - it doesn’t convey any useful meaning on its own (though [reverse mapping](https://www.typescriptlang.org/docs/handbook/enums.html#reverse-mappings) can often help), string enums allow you to give a meaningful and readable value when your code runs, independent of the name of the enum member itself.

## Heterogeneous enums

Technically enums can be mixed with string and numeric members, but it’s not clear why you would ever want to do so:

enum BooleanLikeHeterogeneousEnum {

No = 0,

Yes = "YES",

}[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAQg9nANsAhiAMgSwNbABLAAuwATnAOajBxgDOAouNAN4BQUUAcnFALxQAGADTsoATWC0+UAERj6AZRkiAvkA)

Unless you’re really trying to take advantage of JavaScript’s runtime behavior in a clever way, it’s advised that you don’t do this.

## Computed and constant members

Each enum member has a value associated with it which can be either constant or computed. An enum member is considered constant if:

* It is the first member in the enum and it has no initializer, in which case it’s assigned the value 0:
* // E.X is constant:
* enum E {
* X,

}[Try](https://www.typescriptlang.org/play/#code/PTAEFEDoA1QSwM6gMYHsB2CAuBDdWAuAKAFN0BXAWwlAG8joAaIgXyA)

* It does not have an initializer and the preceding enum member was a numeric constant. In this case the value of the current enum member will be the value of the preceding enum member plus one.
* // All enum members in 'E1' and 'E2' are constant.
* enum E1 {
* X,
* Y,
* Z,
* }
* enum E2 {
* A = 1,
* B,
* C,

}[Try](https://www.typescriptlang.org/play/#code/PTAEEEBtNBTA7ArgW1M2yBGsBOBnUAS3lAHIBRARlNAEN4ATM8gJhtp1lAGMB7ePABd6ggHQAocQhSgqoAN7iAGgBpxATTUAtNQF9J01KwXjwoALyhKagEJqAwnqA)

* The enum member is initialized with a constant enum expression. A constant enum expression is a subset of TypeScript expressions that can be fully evaluated at compile time. An expression is a constant enum expression if it is:
  1. a literal enum expression (basically a string literal or a numeric literal)
  2. a reference to previously defined constant enum member (which can originate from a different enum)
  3. a parenthesized constant enum expression
  4. one of the +, -, ~ unary operators applied to constant enum expression
  5. +, -, \*, /, %, <<, >>, >>>, &, |, ^ binary operators with constant enum expressions as operands

It is a compile time error for constant enum expressions to be evaluated to NaN or Infinity.

In all other cases enum member is considered computed.

enum FileAccess {

// constant members

None,

Read = 1 << 1,

Write = 1 << 2,

ReadWrite = Read | Write,

// computed member

G = "123".length,

}[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAYglgG2AQQMauAZ01A3gKCigHpipUB7ETAFwEMQaoJgIAjYAJ00KgDkqwADS8ASsDoATKAF4oARigAeJQpFEA6pzg1gshctUAmdVHFStOvXPPSAPlEu7TpchQgAHMLukt2XXgBxfQAieSMAZhCAOiQQAHMaAAsRAF8gA)

## Union enums and enum member types

There is a special subset of constant enum members that aren’t calculated: literal enum members. A literal enum member is a constant enum member with no initialized value, or with values that are initialized to

* any string literal (e.g. "foo", "bar, "baz")
* any numeric literal (e.g. 1, 100)
* a unary minus applied to any numeric literal (e.g. -1, -100)

When all members in an enum have literal enum values, some special semantics come to play.

The first is that enum members also become types as well! For example, we can say that certain members can only have the value of an enum member:

enum ShapeKind {

Circle,

Square,

}

interface Circle {

kind: ShapeKind.Circle;

radius: number;

}

interface Square {

kind: ShapeKind.Square;

sideLength: number;

}

let c: Circle = {

kind: ShapeKind.Square,

Type 'ShapeKind.Square' is not assignable to type 'ShapeKind.Circle'.Type 'ShapeKind.Square' is not assignable to type 'ShapeKind.Circle'. radius: 100,

};[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oFCQHYCuAtqAMoAWAhgA6QDSAlvgCagDeOooAwo9AGMANpAA0XcgEdCVaGJwBfHDmYAXGADMqAyL37DdnbgGtmLVJVoMzAOj6CRAbgnQqLRoRSgixAEYxnJRV8dWgtHSkZOQ4JU1YLajomVhsyaVlIZ25ERhZIABkCAHNVClQff2hA5RFVUAFUewNQAF4YkzMEq2SWVPS5cW5Xd09UAEYABgnxBUcgA)

The other change is that enum types themselves effectively become a union of each enum member. With union enums, the type system is able to leverage the fact that it knows the exact set of values that exist in the enum itself. Because of that, TypeScript can catch bugs where we might be comparing values incorrectly. For example:

enum E {

Foo,

Bar,

}

function f(x: E) {

if (x !== E.Foo || x !== E.Bar) {

This condition will always return 'true' since the types 'E.Foo' and 'E.Bar' have no overlap.This condition will always return 'true' since the types 'E.Foo' and 'E.Bar' have no overlap. //

}

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYBsB2AUJAHYCuAtqAKKgDeeooAYnHADR2gBCAhtGwL548AM2KEAxgBcAlnEKghACgAeqCgEoa7KUNDLQAQgC8hygDomcUAB8roJQeNnu0DbXr0Q7AXyA)

In that example, we first checked whether x was not E.Foo. If that check succeeds, then our || will short-circuit, and the body of the ‘if’ will run. However, if the check didn’t succeed, then x can only be E.Foo, so it doesn’t make sense to see whether it’s equal to E.Bar.

## Enums at runtime

Enums are real objects that exist at runtime. For example, the following enum

enum E {

X,

Y,

Z,

}[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAolDeAoKUAaAaZUCamUC1MBfIA)

can actually be passed around to functions

enum E {

X,

Y,

Z,

}

function f(obj: { X: number }) {

return obj.X;

}

// Works, since 'E' has a property named 'X' which is a number.

f(E);[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAolDeAoKUAaAaZUCamUC1MBfRRAMzBAGMAXASwHsQoyAKBgIwCsAuBNPuAgdgAJyhEAlAiyjgNMKOacuAOlQBuRCUQB6XVADqDUQGsAzuijm61YFADkMB1AAWAQ3NR3UAA6iGXzEaAE8oEHcIYAATR1QXAHdXOipXKDovHyERUVVyVhhJDSA)

## Enums at compile time

Even though Enums are real objects that exist at runtime, the keyof keyword works differently than you might expect for typical objects. Instead, use keyof typeof to get a Type that represents all Enum keys as strings.

enum LogLevel {

ERROR,

WARN,

INFO,

DEBUG,

}

/\*\*

\* This is equivalent to:

\* type LogLevelStrings = 'ERROR' | 'WARN' | 'INFO' | 'DEBUG';

\*/

type LogLevelStrings = keyof typeof LogLevel;

function printImportant(key: LogLevelStrings, message: string) {

const num = LogLevel[key];

if (num <= LogLevel.WARN) {

console.log("Log level key is:", key);

console.log("Log level value is:", num);

console.log("Log level message is:", message);

}

}

printImportant("ERROR", "This is a message");[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAMg9gcxsAbsANlA3gKClAUQCUiB5IgGjygHUBBIgOSvwElGAxUlqAEQIBCAVQDiVAL44cAegBUsvLKgAVABYBLAM5QtUYAEcw6lAEN0oAC5QLcAFyLrATwAOwWImRp0AZQsAndRAEbQBeKAByYjIicKgAHwj6JliE8PYuFIj+YRFwgG5FaRwLFzd4JFQMXwCg0KgAa2BHOAAzJ1dW9wqvApwWsBAAYwt1OBAoZxqLVghnOD8LExALAApGx1suzyr-QOCKKAhgTU0TBGBNzV2ggEpsakGxq6hwaDDy7fQAbXWAXQL8Oo2itXlAADzvDyVdAAOiSjDuuHw+EeIE0cHMMPQiBWACJylBzF4Gk0dJpbLiDusbgDkaj0ZjsQg8QSiRgoKZ0GA3FoKQdXjTqCinhjgFicfjEIToYdjqdzmS+bKTmdgIL8JJJJNAtNZvNFss8VFyJSoLi1LpdCZlfLgLiaUA)

### Reverse mappings

In addition to creating an object with property names for members, numeric enums members also get a reverse mapping from enum values to enum names. For example, in this example:

enum Enum {

A,

}

let a = Enum.A;

let nameOfA = Enum[a]; // "A"[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAou0G8BQUoEEA0SC+SkBtgAXKAQygF5Z4A6NAbgOKhFImAHkAzNS6yANqkAuvSgB6cVABEaaUA)

TypeScript compiles this down to the following JavaScript:

"use strict";

var Enum;

(function (Enum) {

Enum[Enum["A"] = 0] = "A";

})(Enum || (Enum = {}));

let a = Enum.A;

let nameOfA = Enum[a]; // "A"[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcAsHsHcCiBbAlgFwFAFMB2BXZURAogb01FAEEAaTAX00wBtt1QBDUAXmNIB01ANyt2oXJ2TYA8gDNqvfoQDanALrDQIUACJquoA)

In this generated code, an enum is compiled into an object that stores both forward (name -> value) and reverse (value -> name) mappings. References to other enum members are always emitted as property accesses and never inlined.

Keep in mind that string enum members do not get a reverse mapping generated at all.

### const enums

In most cases, enums are a perfectly valid solution. However sometimes requirements are tighter. To avoid paying the cost of extra generated code and additional indirection when accessing enum values, it’s possible to use const enums. Const enums are defined using the const modifier on our enums:

const enum Enum {

A = 1,

B = A \* 2,

}[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBApmArgWxgUSag3gKBjAQRgF4YBGAGjxgCETCYAqGAJioF8g)

Const enums can only use constant enum expressions and unlike regular enums they are completely removed during compilation. Const enum members are inlined at use sites. This is possible since const enums cannot have computed members.

const enum Direction {

Up,

Down,

Left,

Right,

}

let directions = [

Direction.Up,

Direction.Down,

Direction.Left,

Direction.Right,

];[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBApmArgWxgEQJYCc7Ch8GAbwCgYYBVABwBoz0QB3MO8gGTgDMpWYAlDAHMAFjxIBfEiQA2cWABNsufOAgwAvDADa9TDjwEwAOmq89yw0bRMWupQfBGO3M-ZXGBIsQF0A3EA)

in generated code will become

"use strict";

let directions = [

0 /\* Up \*/,

1 /\* Down \*/,

2 /\* Left \*/,

3 /\* Right \*/,

];[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcAsHsHcCiBbAlgFwFAGNYDtJ1QBTPAV2VABFUAnY7dVfUAb01FAFUAHAGg7UEeAZwAyxAGbpRoAEqoA5tBmYAvpkwAbYkQAmdBk3yRQAXlABtQTXqNmeAHS9Ztow8dVhrw-fyOJaR87YycFZVUAXQBuIA)

## Ambient enums

Ambient enums are used to describe the shape of already existing enum types.

declare enum Enum {

A = 1,

B,

C = 2,

}[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEIB2BXAtvAoqjBvAUPPAILwC88AjADSHwBCtRAwufAEy0C+QA)

One important difference between ambient and non-ambient enums is that, in regular enums, members that don’t have an initializer will be considered constant if its preceding enum member is considered constant. In contrast, an ambient (and non-const) enum member that does not have initializer is always considered computed.