

TypeScript 3.2 is now available. [Download \(/#download-links\)](#) our latest version today!

Documentation ▼

# Enums

## Enums

Enums allow us 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,  
}
```

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,  
}
```

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 Response {
    No = 0,
    Yes = 1,
}

function respond(recipient: string, message: Response): void {
    // ...
}

respond("Princess Caroline", Response.Yes)
```

Numeric enums can be mixed in computed and constant members (see below). 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, // error! 'A' is not constant-initialized, so 'B' needs an initializer
}
```

## String enums

String enums are a similar concept, but have some subtle runtime differences 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",
}
```

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 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",
}
```

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 }
```

- 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
}
```

- 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
}
```

### 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,
    // ~~~~~ Error!
    radius: 100,
}
```

The other change is that enum types themselves effectively become a *union* of each enum member. While we haven't discussed union types ([./advanced-types.html#union-types](#)) yet, all that you need to know is that 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 silly 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) {
        // ~~~~~
        // Error! Operator '!==' cannot be applied to types 'E.Foo' and 'E.Bar'.
    }
}
```

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 get 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
}
```

can actually be passed around to functions

```
function f(obj: { X: number }) {
    return obj.X;
}

// Works, since 'E' has a property named 'X' which is a number.
f(E);
```

### 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"
```

TypeScript might compile this down to something like the the following JavaScript:

```
var Enum;
(function (Enum) {
    Enum[Enum["A"] = 0] = "A";
})(Enum || (Enum = {}));
var a = Enum.A;
var nameOfA = Enum[a]; // "A"
```

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
}
```

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 Directions {
    Up,
    Down,
    Left,
    Right
}

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right]
```

in generated code will become

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```


## Ambient enums

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

```
declare enum Enum {  
    A = 1,  
    B,  
    C = 2  
}
```

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.

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft