Enums: Named Numeric Constants

An **enum** (short for "enumeration") is a distinct value type that allows you to define a group of named integral constants. Instead of using "magic numbers" (raw numbers with implicit meaning), enums provide more readable and maintainable code by associating names with these numeric values.

For example:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

You can then use this enum type in your code:

```
BorderSide topSide = BorderSide.Top; // Assigning an enum member
bool isTop = (topSide == BorderSide.Top); // Comparing enum values (output: 
true)
```

Underlying Integral Values

Each member of an enum has an **underlying integral value**. By default:

- The underlying type is `int`.
- The constants are automatically assigned values starting from 0, incrementing by 1 for each subsequent member in the declaration order.

So, for `BorderSide`, the default values would be: `Left = 0`, `Right = 1`, `Top = 2`, `Bottom = 3`.

You can **specify an alternative integral type** for the enum, such as `byte`, `short`, `long`, etc.:

```
public enum BorderSide : byte { Left, Right, Top, Bottom } // Underlying type 
is byte
```

You can also **explicitly assign specific underlying values** to enum members:

```
public enum BorderSide : byte { Left = 1, Right = 2, Top = 10, Bottom = 11 }
```

The compiler also allows you to explicitly assign values to only *some* members. Any unassigned members will continue incrementing from the last explicit value.

```
public enum BorderSide : byte
{
    Left = 1,
    Right,   // Automatically assigned 2
    Top = 10,
    Bottom   // Automatically assigned 11
}
```

Enum Conversions

You can convert an enum instance to and from its underlying integral value using an **explicit cast**:

```
int i = (int)BorderSide.Left;    // i will be 1 (assuming Left=1)
BorderSide side = (BorderSide)i; // side will be BorderSide.Left
bool leftOrRight = (int)side <= 2; // Converts 'side' to its int value for
comparison
```

You can also explicitly cast one enum type to another, provided their underlying integral values are compatible. The conversion essentially happens via the underlying numeric values.

```
public enum HorizontalAlignment
{
    Left = BorderSide.Left,  // Assigns 1
    Right = BorderSide.Right, // Assigns 2
    Center                    // Automatically assigned 3
}

HorizontalAlignment h = (HorizontalAlignment)BorderSide.Right; // h will be
HorizontalAlignment.Right
// This is effectively: HorizontalAlignment h =
(HorizontalAlignment)(int)BorderSide.Right;
```

Special Treatment of the Numeric Literal `0`

The numeric literal `0` is treated specially by the compiler in enum expressions and **does not require an explicit cast**:

```
BorderSide b = 0; // OK, no cast needed. 'b' will have the underlying value 0.
if (b == 0) { /* ... */ } // OK
```

This special treatment is due to two common conventions:

- The first member of an enum (which defaults to 0 if unassigned) is often used as a "default" or "none" value.
- For flag enums (discussed next), `0` typically represents "no flags."

Flags Enums

Flags enums (also known as bit field enums) allow you to combine multiple enum members using bitwise operators. This is particularly useful for representing a set of options or permissions.

To prevent ambiguities, members of a combinable enum **must have explicitly assigned values**, typically in powers of two. This ensures that each combination results in a unique underlying integral value.

```
[Flags] // The [Flags] attribute is crucial for proper ToString() behavior
enum BorderSides
{
    None = 0,
    Left = 1,       // 0001
    Right = 2,      // 0010
    Top = 4,        // 0100
    Bottom = 8      // 1000
}
// Or using bit shifts for readability:
// enum BorderSides { None=0, Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3 }
```

You use **bitwise operators** (`|` for OR, `&` for AND, `^` for XOR, `~` for NOT) to combine and test these enum values. These operations act on the underlying integral values.

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right; // Combines Left
and Right (1 | 2 = 3)

if ((leftRight & BorderSides.Left) != 0) // Check if 'Left' is included
    Console.WriteLine("Includes Left"); // Output: Includes Left

string formatted = leftRight.ToString(); // Output: "Left, Right" (thanks to
[Flags] attribute)

BorderSides s = BorderSides.Left;
s |= BorderSides.Right; // Add Right to s (s now holds Left | Right)
Console.WriteLine(s == leftRight); // Output: True

s ^= BorderSides.Right; // Toggle BorderSides.Right (XOR with Right)
Console.WriteLine(s); // Output: Left (Right bit is now off)
```

Convention:

- Always apply the [Flags] attribute to an enum type when its members are combinable. Without it, ToString() on a combined value will just emit the numeric value (e.g., "3") instead of the named members ("Left, Right").
- Combinable enum types are conventionally given a **plural name** (e.g., BorderSides).

For convenience, you can include common combinations as members within the enum declaration itself:

```
[Flags]
enum BorderSides
{
    None = 0,
    Left = 1, Right = 1 << 1, Top = 1 << 2, Bottom = 1 << 3,
    LeftRight = Left | Right,
    TopBottom = Top | Bottom,
    All = LeftRight | TopBottom
}
```

Enum Operators

Most standard operators can be used with enums. They generally operate on the enum's underlying integral values:

- **Assignment:** =
- **Equality/Inequality:** ==, !=
- **Comparison:** <, >, <=, >=
- **Arithmetic:** +, -, ++, -- (additions/subtractions between an enum and an integral type are allowed, but not between two enums).
- **Bitwise:** ^ (XOR), & (AND), | (OR), ~ (NOT)
- **Compound Assignment:** +=, -=
- **Size:** sizeof (returns the size of the underlying type)

The bitwise, arithmetic, and comparison operators will perform their operations on the underlying integral values of the enum.

Type-Safety Issues

While enums enhance type safety, it's important to be aware of some potential issues stemming from their close relationship with integral types.

Because an enum can be explicitly cast to and from its underlying integral type, it's possible to assign a numeric value that does not correspond to any defined enum member:

```
public enum BorderSide { Left, Right, Top, Bottom }
BorderSide b = (BorderSide)12345; // No compile-time error
Console.WriteLine(b); // Output: 12345 (just prints the numeric value)
```

Similarly, bitwise or arithmetic operators can also produce "invalid" enum values.

```
This can lead to issues if your code assumes that an enum variable will always
hold a valid, named member. For example:
void Draw(BorderSide side)
{
    if (side == BorderSide.Left) { /* ... */ }
    else if (side == BorderSide.Right) { /* ... */ }
    else if (side == BorderSide.Top) { /* ... */ }
    else                               { /* ... */ } // Assumes
BorderSide.Bottom, but could be 12345!
}
```

To address this, you should:

- **Add defensive checks:** Include an else clause to handle unexpected values, potentially throwing an ArgumentException.

- **Validate the enum value:** The static Enum.IsDefined method can check if a given integral value corresponds to a named member in a *non-flags* enum.

  ```
  BorderSide side = (BorderSide)12345;
  Console.WriteLine(Enum.IsDefined(typeof(BorderSide), side)); // Output:
  False
  ```

- However, Enum.IsDefined does **not** work correctly for [Flags] enums because a combined flag value (e.g., Left | Right) is not considered a "defined" single member. For flags enums, a common trick is to use Enum.ToString() and decimal.TryParse() as a heuristic to check validity, as shown in the example provided in your document.

Enums are a valuable tool for creating clear, readable, and type-safe code when dealing with fixed sets of choices. Understanding their underlying integral nature and potential type-safety pitfalls allows for their effective and robust use.