1. Checked and Unchecked Contexts

Understanding Integer Overflow

In C#, numeric types have **minimum** and **maximum** values. When you perform arithmetic operations that exceed these limits, **integer overflow** occurs.

Default Behavior (Unchecked Context)

By default, C# operates in an **unchecked context**, which means:

- Overflow exceptions are **NOT thrown**
- Values **wrap around** when they exceed limits
- Silent failures can lead to bugs

Example from code:

```
uint x = uint.MaxValue;  // x = 4,294,967,295
x = x + 10;              // Overflow! Wraps around
Console.WriteLine(x);    // Output: 9
```

Why this happens:

```
uint.MaxValue = 4,294,967,295 (binary: 11111111111111111111111111111111)
            + 10
            = 4,294,967,305 (too large for uint - 32 bits)
Wraps to:     9           (binary: 00000000000000000000000000001001)
Wrap To 9 Because first number is Zero
```

The `checked` Keyword

The `checked` keyword enables **overflow checking**, causing an `OverflowException` when overflow occurs.

Checked Statement Block:

```
checked
{
    int x = int.MaxValue;
    x = x + 10;  // Throws OverflowException
}
```

Checked Expression:

```
uint x = uint.MaxValue;
x = checked(x + 10);  // Throws OverflowException
```

The `unchecked` Keyword

Forces unchecked behavior even if compiler settings enable checking globally.

Example from code:

```
checked
{
    uint x = uint.MaxValue;
    unchecked
    {
        x = x + 10;  // No exception, wraps around
    }
    Console.WriteLine($"x={x}");  // Output: x=9
}
```

When to Use Each:

Context	Use When	Behavior
`checked`	Financial calculations, critical arithmetic	Throws exception on overflow
`unchecked`	Performance-critical code, intentional wraparound	Silent wraparound
Default	General code	Unchecked (wraparound)

Compiler Settings

You can enable checked arithmetic globally:

```
<!-- In .csproj file -->
<PropertyGroup>
    <CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
</PropertyGroup>
```

Performance Considerations

- **Checked operations are slower** (5-20% overhead)
- Use checked for critical calculations only
- Profile before optimizing

Integer Types and Their Limits

```
Console.WriteLine($"byte: {byte.MinValue} to {byte.MaxValue}");
// byte: 0 to 255

Console.WriteLine($"sbyte: {sbyte.MinValue} to {sbyte.MaxValue}");
// sbyte: -128 to 127

Console.WriteLine($"short: {short.MinValue} to {short.MaxValue}");
// short: -32768 to 32767

Console.WriteLine($"ushort: {ushort.MinValue} to {ushort.MaxValue}");
// ushort: 0 to 65535

Console.WriteLine($"int: {int.MinValue} to {int.MaxValue}");
// int: -2147483648 to 2147483647

Console.WriteLine($"uint: {uint.MinValue} to {uint.MaxValue}");
// uint: 0 to 4294967295

Console.WriteLine($"long: {long.MinValue} to {long.MaxValue}");
// long: -9223372036854775808 to 9223372036854775807

Console.WriteLine($"ulong: {ulong.MinValue} to {ulong.MaxValue}");
// ulong: 0 to 18446744073709551615
```

---

2. Parse vs TryParse

Both methods convert strings to numeric types, but they handle errors **very differently**.

`Parse` Method

Throws an **exception** if conversion fails.

```
string input = "123abc";
int x = int.Parse(input);  // Throws FormatException
```

Characteristics:

- Throws `FormatException` for invalid format
- Throws `OverflowException` for out-of-range values
- Requires try-catch for error handling
- **Less performant** due to exception handling

When to use Parse:

- When you're **certain** the input is valid
- When you want exceptions to propagate
- In simple scenarios where exceptions are acceptable

`TryParse` Method

Returns a **boolean** indicating success/failure, outputs the parsed value.

Example from code:

```
Console.WriteLine("enter number");
bool status = int.TryParse(Console.ReadLine(), out int x);
Console.WriteLine($"status={status}, x={x}");
```

Signature:

```
public static bool TryParse(string s, out int result)
```

Characteristics:

- Returns `true` if successful, `false` if failed
- Outputs parsed value via `out` parameter
- Sets output to `0` (default) if parsing fails
- **No exceptions thrown**
- **Better performance** for expected failures

Comparison Table:

Feature	Parse	TryParse
Returns	Parsed value	bool (success/failure)
On failure	Throws exception	Returns false
Output	Direct return	`out` parameter
Performance	Slower (exceptions)	Faster (no exceptions)

Feature	Parse	TryParse
Use case	Guaranteed valid input	User input, uncertain data

Practical Example: Input Validation Loop

Example from code:

```
int x;
bool status = false;
do
{
    Console.Clear();
    Console.WriteLine("enter number");
    status = int.TryParse(Console.ReadLine(), out x);
}
while (status == false);
// Now x contains a valid integer
```

Why TryParse is better here:

- User input is unpredictable
- No exception overhead for each invalid input
- Cleaner code without try-catch
- Better user experience (loop until valid)

Advanced TryParse with Culture

```
using System.Globalization;

string price = "1,234.56";
bool success = decimal.TryParse(
    price,
    NumberStyles.Currency,
    CultureInfo.InvariantCulture,
    out decimal result
);
```

TryParse for Other Types

```
// DateTime
DateTime.TryParse("2026-01-03", out DateTime date);
```

```
// Enum
Enum.TryParse<DayOfWeek>("Monday", out DayOfWeek day);

// Double
double.TryParse("123.45", out double value);

// Boolean
bool.TryParse("true", out bool flag);

// Guid
Guid.TryParse("550e8400-e29b-41d4-a716-446655440000", out Guid id);
```

Best Practices

✅ **Use TryParse for:**

- User input
- File/database data
- Network responses
- Any uncertain data source

✅ **Use Parse for:**

- Hardcoded strings (constants)
- Configuration values you control
- When you want exceptions to bubble up

---

3. Type Checking: `is` and `as` Operators

The `is` Operator

Checks if an object is **compatible** with a given type, returns `bool`.

Basic Type Checking:

```
object obj = 123;
if (obj is int)
{
    Console.WriteLine("obj is an integer");
}
```

Pattern Matching with `is` (C# 7.0+):

Example from code:

```
object obj = 123;
if (obj is int x)  // Pattern matching - declares variable x
{
    Console.WriteLine(x);  // x is already cast to int
}
```

What happens:

1. Checks if `obj` is of type `int`
2. If true, casts `obj` to `int` and assigns to `x`
3. `x` is available within the if block scope
4. Combines type checking and casting in one operation

Everything is an Object:

```
int x = 5;
if(x is object)  // Always true - all types derive from object
{
    Console.WriteLine("x is an object");
}
```

Complex Pattern Matching:

Null Check:

```
string text = null;
if (text is null)
{
    Console.WriteLine("text is null");
}
```

Type Pattern with Property Check:

```
if (obj is employee { id: 5 })
{
    Console.WriteLine("Employee with id 5");
}
```

Relational Patterns (C# 9.0+):

```
int age = 25;
if (age is > 18 and < 65)
{
    Console.WriteLine("Working age");
}
```

List Patterns (C# 11.0+):

```
int[] numbers = { 1, 2, 3 };
if (numbers is [1, 2, 3])
{
    Console.WriteLine("Exact match");
}
```

The `as` Operator

Attempts to **cast** an object to a specified type. Returns `null` if the cast fails (no exception).

Syntax:

```
TargetType variable = object as TargetType;
```

Example from code:

```
object obj = new employee() { id = 12 };
obj = 123;
employee em = obj as employee;  // em will be null (obj is int, not employee)
```

Characteristics:

- Returns `null` on failure (never throws exception)
- Only works with **reference types** and **nullable value types**
- More efficient than casting when failure is possible
- Requires null check after use

`as` vs Direct Casting:

```
// Direct casting - throws InvalidCastException on failure
object obj = 123;
employee em1 = (employee)obj;  // ❌ Throws exception

// Using 'as' - returns null on failure
```

```
employee em2 = obj as employee;  // ✅ em2 is null, no exception
if (em2 != null)
{
    // Safe to use em2
}
```

When to Use `is` vs `as`:

Scenario	Use	Reason
Check type only	`is`	Clearer intent
Check and use	`is` with pattern	One-step operation
Cast with null check	`as`	Avoids exception
Guaranteed type	Direct cast `(Type)`	Clearer intent, fails fast

Combining `as` with Null Conditional:

Example from code (commented):

```
employee em = obj as employee;
if (em == null)
    return false;
else
    return (id == em.id);
```

Modern approach:

```
employee em = obj as employee;
return em?.id == id;  // Using null conditional operator
```

---

4. Overriding `Equals()` Method

The `Equals()` method determines if two objects are **equal**. By default, it checks **reference equality** (same memory location).

Default Behavior (Reference Equality):

```
employee em1 = new employee() { id = 4 };
employee em2 = new employee() { id = 4 };
```

```
Console.WriteLine(em1.Equals(em2));  // False - different objects
Console.WriteLine(em1 == em2);       // False - different references
```

Why false?

- `em1` and `em2` are separate objects in memory
- Default `Equals()` compares memory addresses
- Even though properties are identical, objects are different

Overriding for Value Equality:

Example from code (commented):

```
public override bool Equals(object obj)
{
    // Using 'as' operator
    employee em = obj as employee;
    if (em == null)
        return false;
    else
        return (id == em.id);
}
```

Alternative using `is` operator:

```
public override bool Equals(object obj)
{
    if (obj is employee em)
    {
        return (id == em.id);
    }
    else
        return false;
}
```

Comprehensive `Equals()` Implementation:

```
public override bool Equals(object obj)
{
    // 1. Check for null
    if (obj == null) return false;
```

```
    // 2. Check if same reference
    if (ReferenceEquals(this, obj)) return true;

    // 3. Check if same type
    if (obj.GetType() != this.GetType()) return false;

    // 4. Cast and compare
    employee other = (employee)obj;
    return id == other.id &&
           name == other.name &&
           age == other.age;
}
```

Best Practices for `Equals()`:

1. Override `GetHashCode()` Too:

When you override `Equals()`, you **must** override `GetHashCode()`:

```
public override int GetHashCode()
{
    return HashCode.Combine(id, name, age);
}
```

Why? Collections like `Dictionary` and `HashSet` use hash codes for performance.

2. Implement `IEquatable<T>`: **Not Explain yet !!!!!!!!!!!!**

For better performance and type safety:

```
public class employee : IEquatable<employee>
{
    public int id { get; set; }
    public string name { get; set; }
    public int age { get; set; }

    public bool Equals(employee other)
    {
        if (other == null) return false;
        return id == other.id &&
               name == other.name &&
               age == other.age;
    }

    public override bool Equals(object obj)
```

```
    {
        return Equals(obj as employee);
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(id, name, age);
    }
}
```

3. Overload `==` and `!=` Operators:

```
public static bool operator ==(employee left, employee right)
{
    if (left is null)
        return right is null;
    return left.Equals(right);
}

public static bool operator !=(employee left, employee right)
{
    return !(left == right);
}
```

Equals() Contract Rules:

1. **Reflexive**: `x.Equals(x)` returns `true`
2. **Symmetric**: `x.Equals(y)` returns same as `y.Equals(x)`
3. **Transitive**: If `x.Equals(y)` and `y.Equals(z)`, then `x.Equals(z)`
4. **Consistent**: Multiple calls return same result
5. **Null handling**: `x.Equals(null)` returns `false`

---

5. Nullable Types

Value Types Cannot Be Null (by Default)

```
int x = null;  // ❌ Compile error: Cannot convert null to int
```

Why? Value types (int, double, struct, enum) are stored on the **stack** and must have a value.

Introducing Nullable Value Types

The `Nullable<T>` generic struct allows value types to be null.

Syntax Options:

Full syntax:

```
Nullable<int> x = null;  // ✓ Valid
```

Shorthand (preferred):

```
int? x = null;  // ✓ Valid - syntactic sugar for Nullable<int>
```

Nullable Properties and Methods:

```
int? x = 5;

Console.WriteLine(x.HasValue);    // True
Console.WriteLine(x.Value);       // 5

x = null;
Console.WriteLine(x.HasValue);    // False
Console.WriteLine(x.Value);       // InvalidOperationException!
```

Safe Value Access:

```
int? x = null;

// ❌ Unsafe - throws exception if null
int y = x.Value;

// ✅ Safe - uses default if null
int y = x ?? 0;

// ✅ Safe - check first
if (x.HasValue)
{
    int y = x.Value;
}

// ✅ Safe - GetValueOrDefault
int y = x.GetValueOrDefault();    // Returns 0 if null
int y = x.GetValueOrDefault(10);  // Returns 10 if null
```

Nullable Arithmetic:

```
int? x = 5;
int? y = 10;
int? z = x + y;  // z = 15

int? a = null;
int? b = 5;
int? c = a + b;  // c = null (null propagates)
```

Rule: If any operand is null, the result is null.

Common Nullable Types:

```
int? age = null;
double? price = null;
bool? isActive = null;
DateTime? birthDate = null;
decimal? salary = null;
```

---

6. Nullable Reference Types (C# 8.0+)

By default, reference types (string, classes) can be null. C# 8.0 introduced **nullable reference types** for better null safety.

Enabling Nullable Reference Types:

```
<!-- In .csproj -->
<PropertyGroup>
    <Nullable>enable</Nullable>
</PropertyGroup>
```

Non-nullable vs Nullable Reference Types:

```
// Non-nullable (cannot be null)
string name = "Ali";
name = null;  // ⚠ Warning: Cannot assign null

// Nullable (can be null)
string? description = null;  // ✓ OK
```

Benefits:

1. **Compiler warnings** for potential null reference exceptions
2. **Better documentation** of null expectations
3. **Safer code** by default
4. **IDE support** for null checking

---

7. Null-Conditional Operators

The `?.` Operator (Null-Conditional Member Access)

Safely access members of potentially null objects.

Example from code:

```
int[] arr = { 1, 2, 3 };
int? x = arr?.Length;  // x = 3

arr = null;
int? x = arr?.Length;  // x = null (no NullReferenceException!)
```

Without null-conditional:

```
int? x;
if (arr != null)
    x = arr.Length;
else
    x = null;
```

With null-conditional:

```
int? x = arr?.Length;  // Much cleaner!
```

The `?[]` Operator (Null-Conditional Index Access)

Example from code:

```
int[] arr = { 3, 4, 5 };
Console.WriteLine(arr?[0]);  // Output: 3
```

```
arr = null;
Console.WriteLine(arr?[0]);  // Output: (null, no exception)
```

Chaining Null-Conditional Operators:

```
class Department
{
    public Manager Manager { get; set; }
}

class Manager
{
    public string Name { get; set; }
}

Department dept = null;
string managerName = dept?.Manager?.Name;  // Returns null safely
```

Without null-conditional:

```
string managerName;
if (dept != null && dept.Manager != null)
    managerName = dept.Manager.Name;
else
    managerName = null;
```

Null-Conditional with Method Calls:

```
string text = null;
int? length = text?.ToUpper()?.Length;  // null (no exception)

text = "hello";
int? length = text?.ToUpper()?.Length;  // 5
```

Null-Conditional with Event Invocation: Not Explain Yet

```
// Old way (thread-safe)
EventHandler handler = MyEvent;
if (handler != null)
    handler(this, EventArgs.Empty);
```

```
// New way (much cleaner)
MyEvent?.Invoke(this, EventArgs.Empty);
```

---

8. Null-Coalescing Operators

The `??` Operator (Null-Coalescing)

Returns the left operand if it's not null, otherwise returns the right operand.

Syntax:

```
result = value ?? defaultValue;
```

Example from code:

```
string txt = null;
string txt2 = txt ?? "hi";  // txt2 = "hi"

txt = "hello";
string txt2 = txt ?? "hi";  // txt2 = "hello"
```

Without null-coalescing:

```
string txt2;
if (txt != null)
    txt2 = txt;
else
    txt2 = "hi";
```

Chaining Null-Coalescing:

```
string result = value1 ?? value2 ?? value3 ?? "default";
// Returns first non-null value
```

With Nullable Value Types:

```
int? x = null;
int y = x ?? 0;  // y = 0
```

```
x = 5;
int y = x ?? 0;  // y = 5
```

Common Use Cases:

1. Default Values:

```
string username = GetUsername() ?? "Guest";
int pageSize = GetPageSize() ?? 10;
```

2. Configuration:

```
string connectionString =
    Configuration["ConnectionString"] ??
    "DefaultConnectionString";
```

3. Optional Parameters:

```
public void PrintMessage(string message = null)
{
    Console.WriteLine(message ?? "No message provided");
}
```

The `??=` Operator (Null-Coalescing Assignment) - C# 8.0+

Assigns the right operand only if the left operand is null.

Example from code:

```
string txt = null;
txt ??= "hi";  // txt = "hi"

txt ??= "hello";  // txt still "hi" (not null, so no assignment)
```

Without null-coalescing assignment:

```
if (txt == null)
    txt = "hi";
```

Practical Examples:

Lazy Initialization:

```
private List<string> _items;
public List<string> Items
{
    get
    {
        _items ??= new List<string>();  // Initialize only if null
        return _items;
    }
}
```

Cache Pattern:

```
private string _cachedData;
public string GetData()
{
    _cachedData ??= LoadDataFromDatabase();
    return _cachedData;
}
```

---

9. Ternary (Conditional) Operator

The ternary operator is a **concise if-else** expression.

Syntax:

```
result = condition ? valueIfTrue : valueIfFalse;
```

Example from code:

```
int x = 5;
int y = (x > 5) ? x : 10;  // y = 10 (condition is false)

x = 7;
int y = (x > 5) ? x : 10;  // y = 7 (condition is true)
```

Equivalent if-else:

```
int y;
if (x > 5)
    y = x;
else
    y = 10;
```

Nested Ternary Operators:

```
int score = 85;
string grade = score >= 90 ? "A" :
               score >= 80 ? "B" :
               score >= 70 ? "C" :
               score >= 60 ? "D" : "F";
```

⚠ **Warning:** Nested ternary can be hard to read. Use sparingly.

Common Use Cases:

1. Conditional Assignment:

```
string status = isActive ? "Active" : "Inactive";
```

2. Display Logic:

```
Console.WriteLine($"You have {count} {(count == 1 ? "item" : "items")}");
```

3. Null Handling:

```
string displayName = user?.Name ?? "Anonymous";
```

4. Min/Max:

```
int max = a > b ? a : b;
```

Ternary vs If-Else:

Use	When
Ternary	Simple conditional assignment
If-Else	Complex logic, multiple statements

10. Enumerations (Enums)

Enums define a set of **named constants**, making code more readable and type-safe.

Basic Enum Declaration:

Example from code:

```
enum gender
{
    male,    // 0
    female   // 1
}
```

By default:

- First value is `0`
- Each subsequent value increments by 1
- Underlying type is `int`

Using Enums:

```
gender g = gender.male;
g = gender.female;
Console.WriteLine(g);  // Output: female
```

Enum with Explicit Values:

Example from code:

```
enum orderstatus : byte
{
    pending = 10,    // Explicitly set to 10
    shipped,         // 11 (auto-increment)
    delivered = 100, // Explicitly set to 100
    returned         // 101 (auto-increment)
}
```

Characteristics:

- `: byte` specifies the underlying type (byte instead of int)

- Values can be explicitly assigned
- Unassigned values auto-increment from previous value

Underlying Types:

Enums can use these underlying types:

- `byte` (0 to 255)
- `sbyte` (-128 to 127)
- `short` (-32,768 to 32,767)
- `ushort` (0 to 65,535)
- `int` (default)
- `uint`
- `long`
- `ulong`

Why specify underlying type?

- **Memory optimization**: Use `byte` for small sets
- **Database mapping**: Match column types
- **Interoperability**: Match external system requirements

Converting Between Enum and Numbers:

Enum to Integer:

```
orderstatus o = orderstatus.returned;
int value = (int)o;  // value = 101
```

Integer to Enum:

```
orderstatus o = (orderstatus)107;
Console.WriteLine(o);  // Output: 107 (no validation!)
```

⚠ **Warning:** Casting any integer to enum succeeds, even if value doesn't exist!

Enum Parsing:

`Enum.Parse<T>()` - Throws Exception:

Example from code:

```
orderstatus o1 = Enum.Parse<orderstatus>(Console.ReadLine());
// Input: "pending" → o1 = orderstatus.pending
// Input: "invalid" → FormatException
```

`Enum.TryParse<T>()` - No Exception:

Example from code:

```
bool status = Enum.TryParse<orderstatus>(Console.ReadLine(), out orderstatus
o2);
// Input: "pending" → status = true, o2 = orderstatus.pending
// Input: "invalid" → status = false, o2 = orderstatus.pending (default)
```

Parse Options:

```
// Case-insensitive parsing
Enum.TryParse<orderstatus>("PENDING", true, out orderstatus o);
// true parameter enables case-insensitive
```

Enum Methods:

```
// Get all values: pending,shipped,delivered,returned
orderstatus[] allStatuses = Enum.GetValues<orderstatus>();

// Get all names: pending,shipped,delivered,returned
string[] allNames = Enum.GetNames<orderstatus>();

// Check if value is defined
bool isDefined = Enum.IsDefined(typeof(orderstatus), 10);  // true

// Get name from value
string name = Enum.GetName(typeof(orderstatus), 100);  // "delivered"
```

Enum in Switch Statements:

```
orderstatus status = orderstatus.shipped;

switch (status)
{
    case orderstatus.pending:
        Console.WriteLine("Order is pending");
        break;
```

```
    case orderstatus.shipped:
        Console.WriteLine("Order is shipped");
        break;
    case orderstatus.delivered:
        Console.WriteLine("Order is delivered");
        break;
    case orderstatus.returned:
        Console.WriteLine("Order is returned");
        break;
    default:
        Console.WriteLine("Unknown status");
        break;
}
```

---

11. Flags Enums (Bitwise Enums)

The `[Flags]` attribute indicates an enum represents a **combination of values** (bit flags).

Declaration:

Example from code:

```
[Flags]
enum prev : byte
{
    admin = 1,         // 0001
    instructor = 2,    // 0010
    student = 4,       // 0100
    supervisor = 8,    // 1000
    manager = 16       // 10000
}
```

Key points:

- Values are **powers of 2** (1, 2, 4, 8, 16, 32, ...)
- Each value represents a single bit
- Allows combining multiple values

Why Powers of 2?

```
Binary representation:
admin      = 1   = 00001
```

```
instructor = 2  = 00010
student    = 4  = 00100
supervisor = 8  = 01000
manager    = 16 = 10000

Each occupies a unique bit position!
```

Combining Flags (Bitwise OR - `|`):

```
prev p = prev.admin | prev.student | prev.instructor;
// p = 1 | 4 | 2 = 7 (binary: 00111)
Console.WriteLine(p);  // Output: admin, instructor, student
```

Binary operation:

```
  00001 (admin)
| 00100 (student)
| 00010 (instructor)
-------
  00111 = 7
```

Checking Flags (Bitwise AND - `&`):

```
prev p = prev.admin | prev.student;

if ((p & prev.admin) == prev.admin)
{
    Console.WriteLine("Has admin privilege");  // ✓ Executes
}

if ((p & prev.manager) == prev.manager)
{
    Console.WriteLine("Has manager privilege");  // ✗ Doesn't execute
}
```

Using `HasFlag()` Method:

```
prev p = prev.admin | prev.student;

if (p.HasFlag(prev.admin))
{
```

```
    Console.WriteLine("Has admin privilege");  // ✓ Cleaner syntax
}
```

Adding Flags:

```
prev p = prev.admin;
p |= prev.student;  // Add student privilege
// p now has both admin and student
```

Removing Flags (Bitwise AND NOT):

```
prev p = prev.admin | prev.student | prev.instructor;
p &= ~prev.student;  // Remove student privilege
// p now has only admin and instructor
```

Toggling Flags (Bitwise XOR - `^`):

Example from code:

```
prev p = prev.admin ^ prev.student ^ prev.instructor;
Console.WriteLine(p);  // Output: admin, instructor, student
```

XOR behavior:

```
  00001 (admin)
^ 00100 (student)
^ 00010 (instructor)
-------
  00111 = 7
```

Toggle example:

```
prev p = prev.admin;     // 00001
p ^= prev.admin;         // 00000 (toggle off)
p ^= prev.admin;         // 00001 (toggle on)
```

Practical Flags Example:

```
[Flags]
enum FileAccess
{
```

```
    None = 0,
    Read = 1,
    Write = 2,
    Execute = 4,
    Delete = 8
}

// Grant read and write access
FileAccess access = FileAccess.Read | FileAccess.Write;

// Check permissions
if (access.HasFlag(FileAccess.Write))
{
    // Allow file modification
}

// Add execute permission
access |= FileAccess.Execute;

// Remove write permission
access &= ~FileAccess.Write;

// Check multiple flags
if ((access & (FileAccess.Read | FileAccess.Execute)) == (FileAccess.Read |
FileAccess.Execute))
{
    // Has both read and execute
}
```

Casting Integer to Flags:

Example from code:

```
prev p = (prev)7;
Console.WriteLine(p);  // Output: admin, instructor, student
```

Why 7?

```
7 in binary = 00111
Bit 0 (1)  = admin
Bit 1 (2)  = instructor
Bit 2 (4)  = student
```

Best Practices for Flags:

1. Always Use Powers of 2:

```
[Flags]
enum Permissions
{
    None = 0,
    Read = 1,       // 2^0
    Write = 2,      // 2^1
    Delete = 4,     // 2^2
    Execute = 8,    // 2^3
    Admin = 16      // 2^4
}
```

2. Include a "None" Value:

```
[Flags]
enum Options
{
    None = 0,  // No flags set
    Option1 = 1,
    Option2 = 2,
    Option3 = 4
}
```

3. Include Combinations (Optional):

```
[Flags]
enum FileAccess
{
    None = 0,
    Read = 1,
    Write = 2,
    Execute = 4,
    ReadWrite = Read | Write,       // 3
    FullControl = Read | Write | Execute  // 7
}
```

4. Use Meaningful Names:

```
// ❌ Bad
[Flags]
enum Flags
{
```

```
    Flag1 = 1,
    Flag2 = 2,
    Flag3 = 4
}

// ✅ Good
[Flags]
enum UserPermissions
{
    ViewProfile = 1,
    EditProfile = 2,
    DeleteAccount = 4,
    ManageUsers = 8
}
```

Flags vs Regular Enums:

Feature	Regular Enum	Flags Enum
Purpose	Single value	Multiple values
Values	Any integers	Powers of 2
Combination	Not intended	✓ Intended
`[Flags]` attribute	No	Yes
ToString()	Single name	Comma-separated names
Use case	Status, type	Permissions, options

---

12. Advanced Enum Concepts

Enum Extension Methods:

```
public static class EnumExtensions
{
    public static string GetDescription(this Enum value)
    {
        var field = value.GetType().GetField(value.ToString());
        var attribute = field.GetCustomAttribute<DescriptionAttribute>();
        return attribute?.Description ?? value.ToString();
    }
}
```

```
// Usage:
orderstatus status = orderstatus.pending;
string description = status.GetDescription();
```

Enum with Attributes:

```
using System.ComponentModel;

enum OrderStatus
{
    [Description("Waiting for processing")]
    Pending = 10,

    [Description("Package is on the way")]
    Shipped = 11,

    [Description("Successfully delivered")]
    Delivered = 100,

    [Description("Customer returned the order")]
    Returned = 101
}
```

Enum Validation:

```
public static bool IsValid<T>(int value) where T : Enum
{
    return Enum.IsDefined(typeof(T), value);
}

// Usage:
int input = 107;
if (IsValid<orderstatus>(input))
{
    orderstatus status = (orderstatus)input;
}
else
{
    Console.WriteLine("Invalid status code");
}
```

Enum Iteration:

```
// Iterate all enum values
foreach (orderstatus status in Enum.GetValues<orderstatus>())
{
    Console.WriteLine($"{status} = {(int)status}");
}

// Output:
// pending = 10
// shipped = 11
// delivered = 100
// returned = 101
```

Enum in LINQ:

```
// Get all enum values greater than 50
var highValues = Enum.GetValues<orderstatus>()
    .Cast<orderstatus>()
    .Where(s => (int)s > 50)
    .ToList();
```

---

13. Putting It All Together - Real-World Example

E-commerce Order System:

```
[Flags]
enum ShippingOptions : byte
{
    None = 0,
    Standard = 1,
    Express = 2,
    Overnight = 4,
    International = 8,
    Tracking = 16,
    Insurance = 32,
    SignatureRequired = 64
}

enum OrderStatus : byte
{
    Draft = 0,
    Pending = 10,
    PaymentConfirmed = 20,
```

```
    Processing = 30,
    Shipped = 40,
    Delivered = 50,
    Completed = 60,
    Cancelled = 70,
    Refunded = 80
}

class Order
{
    public int Id { get; set; }
    public decimal? Amount { get; set; }
    public OrderStatus Status { get; set; }
    public ShippingOptions Shipping { get; set; }
    public DateTime? ShippedDate { get; set; }

    public string GetStatusMessage()
    {
        return Status switch
        {
            OrderStatus.Draft => "Order is being prepared",
            OrderStatus.Pending => "Waiting for payment",
            OrderStatus.PaymentConfirmed => "Payment received",
            OrderStatus.Processing => "Processing your order",
            OrderStatus.Shipped => $"Shipped on
{ShippedDate?.ToString("MM/dd/yyyy") ?? "N/A"}",
            OrderStatus.Delivered => "Successfully delivered",
            OrderStatus.Completed => "Order completed",
            OrderStatus.Cancelled => "Order cancelled",
            OrderStatus.Refunded => "Amount refunded",
            _ => "Unknown status"
        };
    }

    public decimal GetShippingCost()
    {
        decimal cost = 0;

        if (Shipping.HasFlag(ShippingOptions.Standard))
            cost += 5.00m;
        if (Shipping.HasFlag(ShippingOptions.Express))
            cost += 15.00m;
        if (Shipping.HasFlag(ShippingOptions.Overnight))
            cost += 30.00m;
        if (Shipping.HasFlag(ShippingOptions.International))
            cost += 50.00m;
```

```
        if (Shipping.HasFlag(ShippingOptions.Tracking))
            cost += 2.50m;
        if (Shipping.HasFlag(ShippingOptions.Insurance))
            cost += (Amount ?? 0) * 0.02m;  // 2% of order amount
        if (Shipping.HasFlag(ShippingOptions.SignatureRequired))
            cost += 3.00m;

        return cost;
    }
}

// Usage:
Order order = new Order
{
    Id = 12345,
    Amount = 250.00m,
    Status = OrderStatus.Pending,
    Shipping = ShippingOptions.Express | ShippingOptions.Tracking |
ShippingOptions.Insurance
};

Console.WriteLine(order.GetStatusMessage());
Console.WriteLine($"Shipping cost: ${order.GetShippingCost():F2}");
```

Key Takeaways Summary

1. **Checked/Unchecked**: Control arithmetic overflow behavior - use `checked` for critical calculations
2. **Parse vs TryParse**: Use `TryParse` for user input (better performance, no exceptions)
3. `is` **operator**: Type checking with pattern matching - combines check and cast
4. `as` **operator**: Safe casting that returns null instead of throwing exceptions
5. `Equals()` **override**: Define custom equality logic for value comparison
6. **Nullable types**: Allow value types to be null using `T?` syntax
7. **Null-conditional** `?.` **and** `?[]`: Safe member and index access without NullReferenceException
8. **Null-coalescing** `??` **and** `??=`: Provide default values for null operands
9. **Ternary operator** `?:`: Concise conditional expressions for simple if-else
10. **Enums**: Named constants for better code readability and type safety
11. **Flags enums**: Combine multiple values using bitwise operations with `[Flags]` attribute
12. **Enum parsing**: Use `Enum.TryParse<T>()` for safe string-to-enum conversion

These concepts are essential for writing **clean**, **safe**, and **maintainable** C# code, especially when dealing with user input, null handling, and domain modeling!

By Abdullah Ali

Contact : +201012613453