Table of Contents

What are Records?

Definition

Records are reference types (or value types with record struct) introduced in C# 9.0, designed specifically for **immutable data models** with **value-based equality**. They provide a concise syntax and automatic implementation of common patterns.

The Problem Records Solve

Before records, creating immutable data classes required a lot of boilerplate code:

```
// Traditional class - LOTS of code needed
public class PersonClass
{
    public string FirstName { get; }
    public string LastName { get; }
    public int Age { get; }

    public PersonClass(string firstName, string lastName, int age)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }

    // Custom equality
    public override bool Equals(object obj)
    {
        if (obj is PersonClass other)
        {
            return FirstName == other.FirstName &&
                   LastName == other.LastName &&
                   Age == other.Age;
        }
        return false;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(FirstName, LastName, Age);
    }

    // ToString
    public override string ToString()
    {
        return $"PersonClass {{ FirstName = {FirstName}, LastName =
{LastName}, Age = {Age} }}";
    }

    // Copy with modifications
    public PersonClass WithAge(int newAge)
    {
        return new PersonClass(FirstName, LastName, newAge);
    }

    // Deconstruction
    public void Deconstruct(out string firstName, out string lastName, out int
age)
    {
        firstName = FirstName;
        lastName = LastName;
        age = Age;
    }
}

// That's 50+ lines of code! 😰
```

What Records Provide Automatically

✓ Auto-Generated Features

When you create a record, C# automatically generates:

- ✅ **Constructor** - Initializes all properties
- ✅ **Properties** - Public init-only properties
- ✅ **Deconstructor** - Extract values as tuples
- ✅ **Equals()** - Value-based equality
- ✅ **GetHashCode()** - Consistent hash codes
- ✅ **== and != operators** - Value comparison
- ✅ **ToString()** - Readable string representation
- ✅ **Copy constructor** - For with expressions
- ✅ **IEquatable** - Type-safe equality

Visual Comparison

```
┌─────────────────────────────────────────────────────────────┐
│                       TRADITIONAL CLASS                     │
├─────────────────────────────────────────────────────────────┤
│ • Constructor (manual)                                      │
│ • Properties (manual)                                       │
│ • Equals() override (50+ lines)                             │
│ • GetHashCode() override (10+ lines)                        │
│ • == operator (10+ lines)                                   │
│ • != operator (5+ lines)                                    │
│ • ToString() (10+ lines)                                    │
│ • Copy methods (10+ lines per property)                     │
│ • Deconstruct (5+ lines)                                    │
│                                                             │
│ TOTAL: 100+ lines of boilerplate code 😰                    │
└─────────────────────────────────────────────────────────────┘

                         ↓ BECOMES ↓

┌─────────────────────────────────────────────────────────────┐
│                            RECORD                           │
```

```
┌──────────────────────────────────────────────────────────┐
│ record Person(string FirstName, string LastName, int Age);│
│                                                          │
│ TOTAL: 1 line! Everything auto-generated! 🎉              │
└──────────────────────────────────────────────────────────┘
```

Key Characteristics

ⓘ **Record Properties**

1. Reference Type by Default

```
record Person(string Name);  // Reference type (class)
record struct Point(int X, int Y);  // Value type (struct)
```

2. Immutable by Default

```
record Person(string Name);
var person = new Person("Ali");
// person.Name = "Sara";  // ❌ ERROR: init-only property
```

3. Value Equality

```
var p1 = new Person("Ali");
var p2 = new Person("Ali");
Console.WriteLine(p1 == p2);  // True (value equality)
```

4. Non-Destructive Mutation

```
var p1 = new Person("Ali");
var p2 = p1 with { Name = "Sara" };  // Creates new instance
```

Why Use Records?

✓ **When to Use Records**

✅ **Data Transfer Objects (DTOs)**

- API requests/responses
- Database models

- Message payloads

✅ **Value Objects (DDD)**

- Money, Address, Email
- Immutable domain concepts

✅ **Configuration Objects**

- Application settings
- Connection strings

✅ **Immutable State**

- Redux/state management
- Event sourcing
- Functional programming

❌ **When NOT to Use:**

- Entities with identity (use classes)
- Mutable state required
- Complex behaviors/methods
- Entity Framework entities (usually)

Record Types

C# provides three types of records:

1. Record Class (Default)

```
// Reference type, immutable
record Person(string Name, int Age);

// Or explicitly:
record class Person(string Name, int Age);
```

2. Record Struct

```
// Value type, immutable
record struct Point(int X, int Y);
```

3. Readonly Record Struct

```
// Value type, guaranteed immutable
readonly record struct Point(int X, int Y);
```

Comparison Table

Feature	record (class)	record struct	readonly record struct
Type	Reference	Value	Value
Memory	Heap	Stack	Stack
Null	Can be null	Cannot be null	Cannot be null
Mutability	Immutable (init)	Mutable by default	Immutable (readonly)
Copying	Reference copy	Value copy	Value copy
Performance	Slower (heap)	Faster (stack)	Fastest (no defensive copy)
Size	Any size	Keep small	Keep small

Examples

```
// Record class - for larger data
record Person(string Name, string Email, string Address, DateTime BirthDate);

// Record struct - for small data
record struct Point(int X, int Y);
record struct Color(byte R, byte G, byte B);

// Readonly record struct - for guaranteed immutability
readonly record struct Vector3(float X, float Y, float Z);
```

Positional Records

Definition

Positional records use a concise syntax where properties are declared in the record declaration itself.

Basic Syntax

```
// Positional record declaration
public record Person(string FirstName, string LastName, int Age);

// Creates:
// - Constructor: Person(string firstName, string lastName, int age)
// - Properties: FirstName, LastName, Age (all init-only)
// - Deconstructor: void Deconstruct(out string, out string, out int)
```

Usage

```
record Person(string FirstName, string LastName, int Age);

// Creating instances
var person1 = new Person("Ali", "Ahmed", 25);
var person2 = new Person(
    FirstName: "Sara",
    LastName: "Mohamed",
    Age: 30
);

// Accessing properties
Console.WriteLine(person1.FirstName);  // Ali
Console.WriteLine(person1.Age);        // 25

// Deconstruction
var (firstName, lastName, age) = person1;
Console.WriteLine(firstName);  // Ali
```

With Default Values

You **cannot** set default values directly in positional syntax, but you can use additional constructors:

```
record Person(string FirstName, string LastName, int Age)
{
    // Additional constructor with defaults
    public Person(string firstName, string lastName)
```

```
        : this(firstName, lastName, 18) { }

    public Person(string firstName)
        : this(firstName, "Unknown", 18) { }
}

// Usage
var p1 = new Person("Ali", "Ahmed", 25);
var p2 = new Person("Sara", "Mohamed");  // Age = 18
var p3 = new Person("Omar");             // LastName = "Unknown", Age = 18
```

Positional with Additional Members

```
record Person(string FirstName, string LastName, int Age)
{
    // Additional properties
    public string Email { get; init; }

    // Computed property
    public string FullName => $"{FirstName} {LastName}";

    // Method
    public bool IsAdult() => Age >= 18;

    // Custom validation
    public string FirstName { get; init; } =
        !string.IsNullOrWhiteSpace(FirstName)
            ? FirstName
            : throw new ArgumentException("FirstName cannot be empty");
}

// Usage
var person = new Person("Ali", "Ahmed", 25)
{
    Email = "ali@example.com"
};

Console.WriteLine(person.FullName);    // Ali Ahmed
Console.WriteLine(person.IsAdult());   // True
```

Multiple Positional Records

```
// Simple records
record Point(int X, int Y);
record Size(int Width, int Height);
record Rectangle(Point Location, Size Dimensions);

// Usage
var rect = new Rectangle(
    new Point(10, 20),
    new Size(100, 50)
);

Console.WriteLine(rect.Location.X);       // 10
Console.WriteLine(rect.Dimensions.Width); // 100

// Deconstruction
var (location, dimensions) = rect;
var (x, y) = location;
var (width, height) = dimensions;
```

Positional Record with Validation

```
record Email(string Address)
{
    // Validate in property
    public string Address { get; init; } = IsValidEmail(Address)
        ? Address
        : throw new ArgumentException("Invalid email format");

    private static bool IsValidEmail(string email)
    {
        return email.Contains("@") && email.Contains(".");
    }
}

// Usage
var email1 = new Email("ali@example.com");  // ✅ OK
// var email2 = new Email("invalid");       // ❌ Exception
```

Nominal Records

Definition

Nominal records use traditional class-like syntax with explicit property declarations.

Basic Syntax

```csharp
public record Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public int Age { get; init; }
}

// Usage
var person = new Person
{
    FirstName = "Ali",
    LastName = "Ahmed",
    Age = 25
};
```

When to Use Nominal vs Positional

Choosing Between Styles

Use Positional Records:

- ✅ Simple data with few properties (2-5)
- ✅ All properties required
- ✅ Want concise syntax
- ✅ Deconstruction is useful

Use Nominal Records:

- ✅ Many properties (6+)
- ✅ Some properties optional
- ✅ Need property initializers
- ✅ Complex validation logic

Nominal with Default Values

```csharp
record AppSettings
{
    public string Host { get; init; } = "localhost";
    public int Port { get; init; } = 8080;
    public bool UseSsl { get; init; } = false;
    public int Timeout { get; init; } = 30;
}

// Usage
var settings1 = new AppSettings();  // All defaults

var settings2 = new AppSettings
{
    Host = "api.example.com",
    Port = 443,
    UseSsl = true
    // Timeout uses default
};
```

Nominal with Constructor

```
record Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public int Age { get; init; }
    public string Email { get; init; }

    // Constructor with validation
    public Person(string firstName, string lastName, int age, string email)
    {
        if (string.IsNullOrWhiteSpace(firstName))
            throw new ArgumentException("FirstName required");

        if (age < 0 || age > 150)
            throw new ArgumentException("Invalid age");

        FirstName = firstName;
        LastName = lastName;
        Age = age;
        Email = email;
    }

    // Parameterless constructor
    public Person() { }
}
```

Nominal with Computed Properties

```
record Employee
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public decimal Salary { get; init; }
    public DateTime HireDate { get; init; }

    // Computed properties
    public string FullName => $"{FirstName} {LastName}";
    public int YearsOfService => DateTime.Now.Year - HireDate.Year;
    public decimal AnnualBonus => Salary * 0.1m;
}
```

Mixing Positional and Nominal

```
// You can mix both styles!
record Person(string FirstName, string LastName)  // Positional
{
    // Additional nominal properties
    public int Age { get; init; }
    public string Email { get; init; }

    // Computed property
    public string FullName => $"{FirstName} {LastName}";
}

// Usage
var person = new Person("Ali", "Ahmed")
{
    Age = 25,
    Email = "ali@example.com"
};
```

Record Properties

Init-Only Properties

Default Immutability

Record properties are **init-only** by default, meaning they can only be set during object initialization.

```
record Person(string Name, int Age);

var person = new Person("Ali", 25);

// ❌ Cannot modify after creation
// person.Name = "Sara";  // ERROR: Init-only property

// ✅ Can set during initialization
var person2 = new Person("Sara", 30)
{
    // If there were additional properties:
    // Email = "sara@example.com"
};
```

Get-Only Properties

```
record Rectangle(double Width, double Height)
{
    // Computed property (get-only)
    public double Area => Width * Height;
    public double Perimeter => 2 * (Width + Height);
}

var rect = new Rectangle(10, 20);
Console.WriteLine(rect.Area);       // 200
Console.WriteLine(rect.Perimeter);  // 60
```

Mutable Properties (Not Recommended)

⚠ Breaking Immutability

You CAN make record properties mutable, but it defeats the purpose of records!

```
record Person(string Name, int Age)
{
    // Override with mutable property
    public string Name { get; set; } = Name;
```

```csharp
    public int Age { get; set; } = Age;
}

var person = new Person("Ali", 25);
person.Age = 26;  // ❌ Works but not recommended!
```

Property Validation

```csharp
record BankAccount(string AccountNumber, decimal Balance)
{
    // Validate in property
    public string AccountNumber { get; init; } =
        !string.IsNullOrWhiteSpace(AccountNumber)
            ? AccountNumber
            : throw new ArgumentException("Account number required");

    public decimal Balance { get; init; } =
        Balance >= 0
            ? Balance
            : throw new ArgumentException("Balance cannot be negative");
}

// Usage
var account = new BankAccount("123456", 1000m);  // ✅ OK
// var invalid = new BankAccount("", -500m);     // ❌ Exception
```

Required Properties (C# 11)

```csharp
record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public int Age { get; init; }  // Optional
}

// Usage
var person = new Person
{
    FirstName = "Ali",
    LastName = "Ahmed"
    // Age is optional
};
```

```
// ❌ ERROR: Missing required properties
// var invalid = new Person { FirstName = "Ali" };
```

Property with Backing Field

```
record Person(string Name, int Age)
{
    private string _email;

    public string Email
    {
        get => _email;
        init
        {
            if (!value.Contains("@"))
                throw new ArgumentException("Invalid email");
            _email = value;
        }
    }
}
```

Value Equality

Definition

Records implement **value equality** (also called structural equality), meaning two records are equal if all their property values are equal, regardless of whether they're the same object instance.

Reference Equality vs Value Equality

Classes (Reference Equality)

```
class PersonClass
{
    public string Name { get; set; }
    public int Age { get; set; }
}

var p1 = new PersonClass { Name = "Ali", Age = 25 };
var p2 = new PersonClass { Name = "Ali", Age = 25 };

Console.WriteLine(p1 == p2);                  // ❌ False (different objects)
Console.WriteLine(p1.Equals(p2));             // ❌ False
Console.WriteLine(object.ReferenceEquals(p1, p2));  // ❌ False
```

Records (Value Equality)

```
record PersonRecord(string Name, int Age);

var p1 = new PersonRecord("Ali", 25);
var p2 = new PersonRecord("Ali", 25);

Console.WriteLine(p1 == p2);                  // ✅ True (same values)
Console.WriteLine(p1.Equals(p2));             // ✅ True
Console.WriteLine(object.ReferenceEquals(p1, p2));  // ❌ False (still
different objects)
```

Visual Representation

```
CLASSES (Reference Equality):
┌─────────────────┐           ┌─────────────────┐
│ p1 @ 0x1000     │           │ p2 @ 0x2000     │
│ Name = "Ali"    │           │ Name = "Ali"    │
│ Age = 25        │           │ Age = 25        │
└─────────────────┘           └─────────────────┘
        ↓                             ↓
   Different memory addresses    Different memory addresses
        ↓                             ↓
   p1 == p2  →  FALSE (comparing addresses)


RECORDS (Value Equality):
┌─────────────────┐           ┌─────────────────┐
│ p1 @ 0x1000     │           │ p2 @ 0x2000     │
│ Name = "Ali"    │           │ Name = "Ali"    │
│ Age = 25        │           │ Age = 25        │
└─────────────────┘           └─────────────────┘
        ↓                             ↓
   Different memory addresses    Different memory addresses
        ↓                             ↓
   p1 == p2  →  TRUE (comparing values, not addresses)
```

Equality Methods

Records automatically implement all equality-related methods:

```
record Person(string Name, int Age);

var p1 = new Person("Ali", 25);
var p2 = new Person("Ali", 25);
var p3 = new Person("Sara", 30);

// == operator
Console.WriteLine(p1 == p2);   // True
Console.WriteLine(p1 == p3);   // False

// != operator
Console.WriteLine(p1 != p2);   // False
Console.WriteLine(p1 != p3);   // True

// Equals method
Console.WriteLine(p1.Equals(p2));   // True
Console.WriteLine(p1.Equals(p3));   // False

// GetHashCode
Console.WriteLine(p1.GetHashCode() == p2.GetHashCode());  // True (same
values)
Console.WriteLine(p1.GetHashCode() == p3.GetHashCode());  // False (different
values)
```

Null Handling

```
record Person(string Name, int Age);

Person p1 = new Person("Ali", 25);
Person p2 = null;
Person p3 = null;

Console.WriteLine(p1 == p2);   // False
Console.WriteLine(p2 == p3);   // True (both null)
Console.WriteLine(p1 != p2);   // True
```

Nested Records

⚠ **Shallow Value Equality**

Value equality compares property values, but for reference type properties, it still compares references!

```csharp
record Address(string Street, string City);
record Person(string Name, Address Address);

var addr1 = new Address("Main St", "Cairo");
var addr2 = new Address("Main St", "Cairo");

var p1 = new Person("Ali", addr1);
var p2 = new Person("Ali", addr2);

Console.WriteLine(p1 == p2);  // ✅ True!
// Address is also a record, so value equality applies to nested records too!
```

But with classes:

```csharp
class AddressClass
{
    public string Street { get; set; }
    public string City { get; set; }
}

record Person(string Name, AddressClass Address);

var addr1 = new AddressClass { Street = "Main St", City = "Cairo" };
var addr2 = new AddressClass { Street = "Main St", City = "Cairo" };

var p1 = new Person("Ali", addr1);
var p2 = new Person("Ali", addr2);

Console.WriteLine(p1 == p2);  // ❌ False!
// Address is a class, so reference equality is used for that property
```

Using Records in Collections

HashSet

```csharp
record Person(string Name, int Age);

var set = new HashSet<Person>();
```

```csharp
set.Add(new Person("Ali", 25));
set.Add(new Person("Ali", 25));  // Not added - considered duplicate!

Console.WriteLine(set.Count);  // 1 (not 2)
```

Dictionary Keys

```csharp
record Coordinate(int X, int Y);

var dict = new Dictionary<Coordinate, string>();

dict[new Coordinate(0, 0)] = "Origin";
dict[new Coordinate(1, 1)] = "Point A";
dict[new Coordinate(1, 1)] = "Point B";  // Overwrites Point A

Console.WriteLine(dict.Count);  // 2
Console.WriteLine(dict[new Coordinate(1, 1)]);  // "Point B"
```

LINQ Operations

```csharp
record Product(string Name, decimal Price);

var products = new List<Product>
{
    new Product("Laptop", 1000),
    new Product("Mouse", 50),
    new Product("Laptop", 1000),  // Duplicate
    new Product("Keyboard", 100)
};

// Distinct works with value equality
var unique = products.Distinct();
Console.WriteLine(unique.Count());  // 3 (one duplicate removed)

// Contains works with value equality
bool hasLaptop = products.Contains(new Product("Laptop", 1000));
Console.WriteLine(hasLaptop);  // True
```

Performance Impact

Equality Performance

Value equality is slightly slower than reference equality because it must compare all properties:

```
record LargeRecord(
    string Prop1, string Prop2, string Prop3,
    int Prop4, int Prop5, int Prop6,
    // ... many more properties
    DateTime Prop20
);

// Equality check must compare ALL properties
var r1 = new LargeRecord(/* ... */);
var r2 = new LargeRecord(/* ... */);

// Compares: Prop1 == Prop1 && Prop2 == Prop2 && ... && Prop20 == Prop20
bool equal = r1 == r2;
```

Optimization tip: If you have many properties, consider if you really need value equality, or if reference equality (regular class) would suffice.

With Expression

Definition

The `with` expression creates a **copy** of a record with specified properties modified. This is called **non-destructive mutation**.

Basic Syntax

```
record Person(string Name, int Age, string City);

var person1 = new Person("Ali", 25, "Cairo");

// Create a copy with Age changed
var person2 = person1 with { Age = 26 };

Console.WriteLine(person1);  // Person { Name = Ali, Age = 25, City = Cairo }
Console.WriteLine(person2);  // Person { Name = Ali, Age = 26, City = Cairo }
```

How It Works

```
Original Record:
┌───────────────────────────┐
│ person1                   │
│ Name = "Ali"              │
│ Age = 25                  │  ──────┐  with { Age = 26 }
│ City = "Cairo"            │        │
└───────────────────────────┘        │
        (unchanged)                  │
                                     ↓
New Record (Copy):                Creates
┌───────────────────────────┐        │
│ person2                   │  ←─────┘
│ Name = "Ali"    (copied)  │
│ Age = 26        (changed) │
│ City = "Cairo"  (copied)  │
└───────────────────────────┘
```

Single Property Change

```csharp
record Product(string Name, decimal Price, int Stock);

var product = new Product("Laptop", 1000m, 50);

// Change just the price
var discountedProduct = product with { Price = 800m };

// Change just the stock
var restockedProduct = product with { Stock = 100 };

Console.WriteLine(product);            // Product { Name = Laptop, Price =
1000, Stock = 50 }
Console.WriteLine(discountedProduct);  // Product { Name = Laptop, Price =
800, Stock = 50 }
Console.WriteLine(restockedProduct);   // Product { Name = Laptop, Price =
1000, Stock = 100 }
```

Multiple Property Changes

```csharp
record Employee(string Name, string Department, decimal Salary);

var emp1 = new Employee("Ali", "IT", 50000);
```

```
// Change multiple properties at once
var emp2 = emp1 with
{
    Department = "Management",
    Salary = 80000
};

Console.WriteLine(emp1);  // Employee { Name = Ali, Department = IT, Salary =
50000 }
Console.WriteLine(emp2);  // Employee { Name = Ali, Department = Management,
Salary = 80000 }
```

Chaining With Expressions

```
record Config(string Env, string Host, int Port, bool Debug);

var prodConfig = new Config("production", "api.example.com", 443, false);

// Chain multiple with expressions
var testConfig = prodConfig with { Env = "test" };
var localConfig = testConfig with { Host = "localhost", Port = 8080 };
var debugConfig = localConfig with { Debug = true };

// Or chain directly in one expression
var devConfig = prodConfig with { Env = "development" }
                          with { Host = "localhost" }
                          with { Port = 8080, Debug = true };
```

With Expression on Nested Records

```
record Address(string Street, string City, string Country);
record Person(string Name, int Age, Address Address);

var person = new Person(
    "Ali",
    25,
    new Address("Main St", "Cairo", "Egypt")
);

// Update nested record
var movedPerson = person with
{
    Address = person.Address with { City = "Alexandria" }
};
```

```
Console.WriteLine(person.Address.City);       // Cairo
Console.WriteLine(movedPerson.Address.City);  // Alexandria
```

Practical Example: State Management

```
record GameState(
    int Level,
    int Score,
    int Lives,
    bool IsPaused
);

class Game
{
    private GameState _state;

    public Game()
    {
        _state = new GameState(1, 0, 3, false);
    }

    public void AddPoints(int points)
    {
        _state = _state with { Score = _state.Score + points };
    }

    public void LoseLife()
    {
        _state = _state with { Lives = _state.Lives - 1 };
    }

    public void NextLevel()
    {
        _state = _state with
        {
            Level = _state.Level + 1,
            Score = _state.Score + 1000  // Level bonus
        };
    }

    public void TogglePause()
    {
        _state = _state with { IsPaused = !_state.IsPaused };
    }
}
```

With Expression in Functional Programming

```
record User(string Name, string Email, bool IsActive);

// Pure function - returns new user, doesn't modify input
User ActivateUser(User user) => user with { IsActive = true };
User DeactivateUser(User user) => user with { IsActive = false };
User UpdateEmail(User user, string newEmail) => user with { Email = newEmail
};

// Usage
var user = new User("Ali", "ali@old.com", false);
var activeUser = ActivateUser(user);
var updatedUser = UpdateEmail(activeUser, "ali@new.com");

// Original unchanged
Console.WriteLine(user.Email);         // ali@old.com
Console.WriteLine(user.IsActive);      // false

// New versions
Console.WriteLine(updatedUser.Email);    // ali@new.com
Console.WriteLine(updatedUser.IsActive); // true
```

Performance Considerations

⚠ Memory Allocation

Each `with` expression creates a **new object**:

```
var person = new Person("Ali", 25, "Cairo");

// Each creates NEW object
var p2 = person with { Age = 26 };      // New allocation
var p3 = p2 with { Age = 27 };          // New allocation
var p4 = p3 with { City = "Alex" };     // New allocation
```

For frequent updates in tight loops, this can impact performance. Consider:

- Using mutable types for hot paths
- Batching updates when possible

- Using structs for small data

Deconstruction

Definition

Deconstruction allows you to "unpack" a record into individual variables using tuple syntax.

Automatic Deconstruction

Records automatically provide deconstruction for positional parameters:

```
record Person(string Name, int Age, string City);

var person = new Person("Ali", 25, "Cairo");

// Deconstruct into variables
var (name, age, city) = person;

Console.WriteLine(name);   // Ali
Console.WriteLine(age);    // 25
Console.WriteLine(city);   // Cairo
```

How Deconstruction Works

When you create a positional record, the compiler generates a `Deconstruct` method:

```
// You write this:
record Person(string Name, int Age);

// Compiler generates this:
public record Person
{
    public string Name { get; init; }
    public int Age { get; init; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Auto-generated Deconstruct method
    public void Deconstruct(out string name, out int age)
    {
        name = this.Name;
        age = this.Age;
    }
}
```

Deconstruction Patterns

Pattern 1: Extract All Values

```
record Product(string Name, decimal Price, int Stock);

var product = new Product("Laptop", 1000m, 50);
var (name, price, stock) = product;

Console.WriteLine($"{name}: ${price} ({stock} in stock)");
```

Pattern 2: Discard Unwanted Values

```
record Employee(int Id, string Name, string Dept, decimal Salary);

var emp = new Employee(1, "Ali", "IT", 75000);

// Only care about Name and Salary
var (_, name, _, salary) = emp;

Console.WriteLine($"{name} earns ${salary}");
```

Pattern 3: Deconstruction in Foreach

```
record Point(int X, int Y);

var points = new List<Point>
{
    new Point(0, 0),
    new Point(10, 20),
    new Point(30, 40)
};
```

```
foreach (var (x, y) in points)
{
    Console.WriteLine($"X: {x}, Y: {y}");
}
```

Pattern 4: In Switch Expressions

```
record Shape(string Type, double Dimension1, double Dimension2);

double CalculateArea(Shape shape) => shape switch
{
    ("Circle", var radius, _) => Math.PI * radius * radius,
    ("Rectangle", var width, var height) => width * height,
    ("Square", var side, _) => side * side,
    _ => 0
};

var circle = new Shape("Circle", 5, 0);
var rect = new Shape("Rectangle", 10, 20);

Console.WriteLine(CalculateArea(circle));  // ~78.5
Console.WriteLine(CalculateArea(rect));    // 200
```

Pattern 5: In LINQ

```
record Student(int Id, string Name, int Grade);

var students = new List<Student>
{
    new Student(1, "Ali", 85),
    new Student(2, "Sara", 92),
    new Student(3, "Omar", 78)
};

// Deconstruct in LINQ query
var topStudents = students
    .Where(s => s.Grade >= 80)
    .Select(s =>
    {
        var (id, name, grade) = s;
        return $"{name} (ID: {id}) - Grade: {grade}";
    });
```

Custom Deconstruction

You can add additional `Deconstruct` methods with different signatures:

```
record Person(string FirstName, string LastName, int Age, string Email)
{
    // Additional deconstructor - just name parts
    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }

    // Another deconstructor - full name and age
    public void Deconstruct(out string fullName, out int age)
    {
        fullName = $"{FirstName} {LastName}";
        age = Age;
    }
}

var person = new Person("Ali", "Ahmed", 25, "ali@example.com");

// Use different deconstructors
var (first, last) = person;                   // 2 variables
var (fullName, age) = person;                 // Different 2 variables
var (firstName, lastName, age, email) = person; // All 4 variables
```

Deconstruction with Pattern Matching

```
record Temperature(double Value, string Unit);

string Describe(Temperature temp) => temp switch
{
    (< 0, "Celsius") => "Freezing",
    (>= 0 and < 10, "Celsius") => "Cold",
    (>= 10 and < 20, "Celsius") => "Cool",
    (>= 20 and < 30, "Celsius") => "Warm",
    (>= 30, "Celsius") => "Hot",
    var (value, unit) => $"{value}°{unit}"
};

var temp1 = new Temperature(-5, "Celsius");
var temp2 = new Temperature(25, "Celsius");
var temp3 = new Temperature(98.6, "Fahrenheit");
```

```
Console.WriteLine(Describe(temp1));  // Freezing
Console.WriteLine(Describe(temp2));  // Warm
Console.WriteLine(Describe(temp3));  // 98.6°Fahrenheit
```

ToString Implementation

Definition

Records automatically generate a user-friendly `ToString()` implementation that includes the record type name and all property values.

Automatic ToString

```
record Person(string Name, int Age, string City);

var person = new Person("Ali", 25, "Cairo");

Console.WriteLine(person.ToString());
// Output: Person { Name = Ali, Age = 25, City = Cairo }

// ToString is called implicitly
Console.WriteLine(person);
// Output: Person { Name = Ali, Age = 25, City = Cairo }
```

Format

The generated format is:

```
RecordTypeName { Property1 = Value1, Property2 = Value2, ... }
```

Nested Records

```
record Address(string Street, string City);
record Person(string Name, Address Address);

var person = new Person(
    "Ali",
    new Address("Main St", "Cairo")
);
```

```
Console.WriteLine(person);
// Output: Person { Name = Ali, Address = Address { Street = Main St, City =
Cairo } }
```

With Collections

```
record Student(string Name, List<int> Grades);

var student = new Student(
    "Ali",
    new List<int> { 85, 90, 88 }
);

Console.WriteLine(student);
// Output: Student { Name = Ali, Grades =
System.Collections.Generic.List`1[System.Int32] }
// Note: Collections show type name, not contents
```

Custom ToString

You can override the default ToString:

```
record Person(string FirstName, string LastName, int Age)
{
    public override string ToString()
    {
        return $"{FirstName} {LastName} ({Age} years old)";
    }
}

var person = new Person("Ali", "Ahmed", 25);
Console.WriteLine(person);
// Output: Ali Ahmed (25 years old)
```

Using ToString in Logging

```
record ApiRequest(
    string Method,
    string Url,
    DateTime Timestamp,
    Dictionary<string, string> Headers
);
```

```
void LogRequest(ApiRequest request)
{
    // ToString automatically includes all details
    Console.WriteLine($"[{DateTime.Now}] {request}");
}

var request = new ApiRequest(
    "GET",
    "https://api.example.com/users",
    DateTime.Now,
    new Dictionary<string, string> { { "Authorization", "Bearer token" } }
);

LogRequest(request);
// Output: [timestamp] ApiRequest { Method = GET, Url = https://..., Timestamp
= ..., Headers = ... }
```

ToString in Debugging

```
record OrderItem(string ProductName, int Quantity, decimal Price);
record Order(int OrderId, List<OrderItem> Items, decimal Total);

var order = new Order(
    1001,
    new List<OrderItem>
    {
        new OrderItem("Laptop", 1, 1000m),
        new OrderItem("Mouse", 2, 25m)
    },
    1050m
);

// Great for debugging - see all values at once
Console.WriteLine(order);
// Order { OrderId = 1001, Items =
System.Collections.Generic.List`1[OrderItem], Total = 1050 }
```

Inheritance with Records

Definition

Records support inheritance, allowing you to create derived records that inherit properties and behavior from base records.

Basic Inheritance

```
// Base record
record Person(string Name, int Age);

// Derived record
record Employee(string Name, int Age, string Department, decimal Salary)
    : Person(Name, Age);

// Usage
var person = new Person("Ali", 25);
var employee = new Employee("Sara", 30, "IT", 75000);

Console.WriteLine(person);    // Person { Name = Ali, Age = 25 }
Console.WriteLine(employee);  // Employee { Name = Sara, Age = 30, Department
= IT, Salary = 75000 }
```

Inheritance Rules

Record Inheritance Rules

1. **Records can only inherit from records** (not classes)
2. **Classes cannot inherit from records**
3. **Records are sealed by default** (but can be made inheritable)
4. **All positional parameters must be passed to base**

Positional Record Inheritance

```
record Animal(string Name, int Age);
record Dog(string Name, int Age, string Breed) : Animal(Name, Age);
record Cat(string Name, int Age, bool IsIndoor) : Animal(Name, Age);

var dog = new Dog("Buddy", 3, "Golden Retriever");
var cat = new Cat("Whiskers", 2, true);

Console.WriteLine(dog);  // Dog { Name = Buddy, Age = 3, Breed = Golden
Retriever }
Console.WriteLine(cat);  // Cat { Name = Whiskers, Age = 2, IsIndoor = True }
```

Adding Properties in Derived Record

```
record Vehicle(string Make, string Model, int Year);

record Car(string Make, string Model, int Year, int Doors)
    : Vehicle(Make, Model, Year);

record Motorcycle(string Make, string Model, int Year, string Type)
    : Vehicle(Make, Model, Year);

var car = new Car("Toyota", "Camry", 2024, 4);
var bike = new Motorcycle("Honda", "CBR", 2024, "Sport");
```

Adding Methods in Derived Record

```
record Shape(string Type, string Color)
{
    public virtual double CalculateArea() => 0;
}

record Circle(string Color, double Radius)
    : Shape("Circle", Color)
{
    public override double CalculateArea() => Math.PI * Radius * Radius;
}

record Rectangle(string Color, double Width, double Height)
    : Shape("Rectangle", Color)
{
    public override double CalculateArea() => Width * Height;
}

// Usage
Shape circle = new Circle("Red", 5);
Shape rectangle = new Rectangle("Blue", 10, 20);

Console.WriteLine(circle.CalculateArea());     // ~78.54
Console.WriteLine(rectangle.CalculateArea());  // 200
```

Polymorphism with Records

```
record Employee(string Name, decimal BaseSalary)
{
    public virtual decimal CalculateSalary() => BaseSalary;
}

record Manager(string Name, decimal BaseSalary, decimal Bonus)
    : Employee(Name, BaseSalary)
{
    public override decimal CalculateSalary() => BaseSalary + Bonus;
}

record Developer(string Name, decimal BaseSalary, int Projects)
    : Employee(Name, BaseSalary)
{
    public override decimal CalculateSalary() => BaseSalary + (Projects *
1000);
}

// Polymorphic collection
List<Employee> employees = new()
{
    new Employee("John", 50000),
    new Manager("Alice", 80000, 20000),
    new Developer("Bob", 70000, 5)
};

foreach (var emp in employees)
{
    Console.WriteLine($"{emp.Name}: ${emp.CalculateSalary()}");
}
// Output:
// John: $50000
// Alice: $100000
// Bob: $75000
```

Equality with Inheritance

⚠ **Type Checking in Equality**

Two records are only equal if they are of the **same type** and have equal property values:

```
record Person(string Name);
record Employee(string Name, int EmployeeId) : Person(Name);
```

```csharp
var person = new Person("Ali");
var employee = new Employee("Ali", 123);

Console.WriteLine(person == employee);  // False - different types!
Console.WriteLine(person.Name == employee.Name);  // True - same Name value
```

Sealed Records

```csharp
// Sealed record - cannot be inherited
sealed record FinalPerson(string Name, int Age);

// ❌ ERROR: Cannot inherit from sealed record
// record Employee(string Name, int Age, string Dept) : FinalPerson(Name, 
Age);
```

Abstract Records

```csharp
abstract record Shape(string Color)
{
    public abstract double Area { get; }
}

record Circle(string Color, double Radius) : Shape(Color)
{
    public override double Area => Math.PI * Radius * Radius;
}

record Square(string Color, double Side) : Shape(Color)
{
    public override double Area => Side * Side;
}

// Cannot instantiate abstract record
// var shape = new Shape("Red");  // ❌ ERROR

// Can instantiate derived records
Shape circle = new Circle("Red", 5);
Shape square = new Square("Blue", 10);

Console.WriteLine($"Circle area: {circle.Area}");
Console.WriteLine($"Square area: {square.Area}");
```

Record Structs

Definition

Record structs (C# 10) are value-type records that combine the benefits of records (value equality, with expressions) with the performance characteristics of structs (stack allocation).

Basic Record Struct

```
// Record struct - value type
record struct Point(int X, int Y);

var p1 = new Point(10, 20);
var p2 = new Point(10, 20);

Console.WriteLine(p1 == p2);  // True (value equality)
Console.WriteLine(p1.GetType());  // Point (value type)
```

Record Struct vs Record Class

```
// Record class (reference type, heap allocated)
record class PersonClass(string Name, int Age);

// Record struct (value type, stack allocated)
record struct PersonStruct(string Name, int Age);

// Memory allocation
var personClass = new PersonClass("Ali", 25);   // Heap
var personStruct = new PersonStruct("Sara", 30); // Stack

// Copying behavior
var copy1 = personClass;   // Reference copy (same object)
var copy2 = personStruct;  // Value copy (new independent copy)

copy2 = copy2 with { Age = 31 };

// personClass and copy1 point to same object
// personStruct and copy2 are independent
```

Mutable Record Struct

```
// Mutable record struct
record struct MutablePoint(int X, int Y)
{
    public int X { get; set; } = X;
    public int Y { get; set; } = Y;
}

var point = new MutablePoint(10, 20);
point.X = 30;  // ✅ Allowed - mutable
Console.WriteLine(point);  // MutablePoint { X = 30, Y = 20 }
```

Readonly Record Struct

```
// Readonly record struct - fully immutable
readonly record struct ImmutablePoint(int X, int Y);

var point = new ImmutablePoint(10, 20);
// point.X = 30;  // ❌ ERROR: Properties are readonly

// Can only use with expressions
var newPoint = point with { X = 30 };
Console.WriteLine(newPoint);  // ImmutablePoint { X = 30, Y = 20 }
```

Comparison Table

Feature	record class	record struct	readonly record struct
Type	Reference	Value	Value
Allocation	Heap	Stack	Stack
Null	Can be null	Cannot	Cannot
Mutability	Immutable (init)	Mutable by default	Fully immutable
Performance	Slower (heap)	Faster	Fastest (no defensive copy)
Use Case	Large data, shared state	Small data, local scope	Small immutable data

When to Use Record Structs

✓ **Use Record Struct When:**

- ✅ Data is **small** (<= 16 bytes recommended)

- ✅ **Short-lived** objects (local scope)
- ✅ Need **value equality**
- ✅ Need **with expressions**
- ✅ **Performance critical** (avoid heap allocations)

❌ Avoid Record Struct When:

- Data is large (>16 bytes)
- Objects are long-lived
- Need to be nullable frequently
- Passed around a lot (copying overhead)

Examples

Example 1: Small Coordinates

```
readonly record struct Point2D(int X, int Y);
readonly record struct Point3D(int X, int Y, int Z);

// Stack allocated, fast, immutable
var p1 = new Point2D(10, 20);
var p2 = p1 with { X = 30 };

var p3 = new Point3D(1, 2, 3);
var (x, y, z) = p3;  // Deconstruction works
```

Example 2: RGB Color

```
readonly record struct Color(byte R, byte G, byte B)
{
    // Named colors
    public static readonly Color Red = new(255, 0, 0);
    public static readonly Color Green = new(0, 255, 0);
    public static readonly Color Blue = new(0, 0, 255);

    // Computed property
    public string Hex => $"#{R:X2}{G:X2}{B:X2}";
}

var color = Color.Red;
var lighterRed = color with { R = 200 };
Console.WriteLine(lighterRed.Hex);  // #C80000
```

Example 3: Range

```
readonly record struct Range(int Start, int End)
{
    public int Length => End - Start;
    public bool Contains(int value) => value >= Start && value < End;
}

var range = new Range(10, 20);
Console.WriteLine(range.Length);      // 10
Console.WriteLine(range.Contains(15)); // True
```

Mutable vs Immutable Records

Immutable Records (Default)

```
// Properties are init-only by default
record Person(string Name, int Age);

var person = new Person("Ali", 25);
// person.Age = 26;  // ❌ ERROR: Cannot modify init-only property

// Use with expression for changes
var older = person with { Age = 26 };  // ✅ Creates new instance
```

Making Records Mutable

```
// Override properties with set accessors
record MutablePerson(string Name, int Age)
{
    public string Name { get; set; } = Name;
    public int Age { get; set; } = Age;
}

var person = new MutablePerson("Ali", 25);
person.Age = 26;  // ✅ Allowed
Console.WriteLine(person.Age);  // 26
```

Comparison

Aspect	Immutable	Mutable
Thread Safety	✅ Safe	❌ Not safe
Predictability	✅ High	⚠️ Lower
Debugging	✅ Easier	❌ Harder
Performance	⚠️ More allocations	✅ Fewer allocations
Functional Programming	✅ Fits well	❌ Doesn't fit
with Expression	✅ Natural	⚠️ Works but odd

When to Use Each

Choose Immutable When:

- ✅ Thread safety matters
- ✅ Data represents values/facts
- ✅ Functional programming style
- ✅ State management (Redux, etc.)
- ✅ DTOs, API models

Choose Mutable When:

- ✅ Performance critical (tight loops)
- ✅ Builder patterns
- ✅ Temporary working data
- ✅ Entity Framework entities
- ✅ Large objects updated frequently

Records vs Classes vs Structs

Comprehensive Comparison

Feature	Class	Record	Struct	Record Struct
Type	Reference	Reference	Value	Value
Memory	Heap	Heap	Stack	Stack

Feature	Class	Record	Struct	Record Struct
Nullable	Yes	Yes	No (unless Nullable)	No
Equality	Reference	Value	Value	Value
Immutability	Manual	Default (init)	Manual	Default/readonly
with Expression	❌ No	✅ Yes	❌ No	✅ Yes
Deconstruction	Manual	Auto	Manual	Auto
ToString	Manual	Auto	Manual	Auto
Inheritance	✅ Yes	✅ Yes	❌ No	❌ No
Performance	Slower	Slower	Faster	Faster
Use Case	Entities, behaviors	DTOs, values	Small data	Small immutable data

Visual Decision Tree

```
Do you need value equality?
├─ No → Use CLASS
└─ Yes
   ├─ Is data small (<= 16 bytes)?
   │  ├─ Yes → Use RECORD STRUCT (readonly if immutable)
   │  └─ No → Use RECORD
   └─ Do you need inheritance?
      ├─ Yes → Use RECORD or CLASS
      └─ No → Use RECORD or STRUCT
```

Example Scenarios

Scenario 1: Entity with Identity

```
// Use CLASS - entity has identity, not value
class User
{
    public int Id { get; set; }
    public string Username { get; set; }
    public string Email { get; set; }

    // Equality based on ID, not all properties
    public override bool Equals(object obj)
    {
        return obj is User other && Id == other.Id;
    }
}
```

Scenario 2: Data Transfer Object

```
// Use RECORD - value object, immutable
record UserDto(int Id, string Username, string Email);
```

Scenario 3: Small Coordinate

```
// Use RECORD STRUCT - small, value type
readonly record struct Point(int X, int Y);
```

Scenario 4: Complex Domain Model

```
// Use RECORD - immutable value object with behavior
record Money(decimal Amount, string Currency)
{
    public Money Add(Money other)
    {
        if (Currency != other.Currency)
            throw new InvalidOperationException("Currency mismatch");

        return this with { Amount = Amount + other.Amount };
    }

    public static Money operator +(Money left, Money right) =>
left.Add(right);
}
```

Real-World Use Cases

Use Case 1: API Response Models

```
record ApiResponse<T>(
    bool Success,
    T Data,
    string Message,
    List<string> Errors
```

```csharp
);

record UserResponse(
    int Id,
    string Username,
    string Email,
    DateTime CreatedAt
);

// Usage
var response = new ApiResponse<UserResponse>(
    Success: true,
    Data: new UserResponse(1, "ali", "ali@example.com", DateTime.Now),
    Message: "User retrieved successfully",
    Errors: new List<string>()
);
```

Use Case 2: Configuration

```csharp
record DatabaseConfig(
    string Host,
    int Port,
    string Database,
    string Username,
    string Password
)
{
    public string ConnectionString =>
        $"Server={Host};Port={Port};Database={Database};User=
{Username};Password={Password}";
}

record AppConfig(
    DatabaseConfig Database,
    string LogPath,
    int MaxRetries,
    TimeSpan Timeout
);

// Usage
var config = new AppConfig(
    new DatabaseConfig("localhost", 5432, "mydb", "admin", "pass"),
    "/var/log/app.log",
    3,
    TimeSpan.FromSeconds(30)
);

// Easy to create variants
var devConfig = config with
{
    Database = config.Database with { Host = "dev-server" }
};
```

Use Case 3: Event Sourcing

```
abstract record DomainEvent(Guid AggregateId, DateTime Timestamp);

record OrderCreated(
    Guid AggregateId,
    DateTime Timestamp,
    string CustomerName,
    List<OrderItem> Items
) : DomainEvent(AggregateId, Timestamp);

record OrderItem(string ProductId, int Quantity, decimal Price);

record OrderShipped(
    Guid AggregateId,
    DateTime Timestamp,
    string TrackingNumber,
    string Carrier
) : DomainEvent(AggregateId, Timestamp);

record OrderCancelled(
    Guid AggregateId,
    DateTime Timestamp,
    string Reason
) : DomainEvent(AggregateId, Timestamp);

// Event store
class EventStore
{
    private List<DomainEvent> _events = new();

    public void Store(DomainEvent @event)
    {
        _events.Add(@event);
    }

    public IEnumerable<DomainEvent> GetEventsForAggregate(Guid aggregateId)
    {
        return _events.Where(e => e.AggregateId == aggregateId);
    }
}
```

Use Case 4: State Management (Redux-like)

```
record AppState(
    UserState User,
    List<Product> Products,
    ShoppingCart Cart,
    bool IsLoading
);

record UserState(string Username, string Email, bool IsAuthenticated);

record Product(int Id, string Name, decimal Price, int Stock);

record ShoppingCart(List<CartItem> Items, decimal Total);

record CartItem(Product Product, int Quantity);

// Reducers
class AppReducer
{
    public static AppState Reduce(AppState state, object action)
    {
        return action switch
        {
            UserLoginAction login => state with
            {
                User = new UserState(login.Username, login.Email, true)
            },

            AddToCartAction add => state with
            {
                Cart = AddItemToCart(state.Cart, add.Product, add.Quantity)
            },

            SetLoadingAction loading => state with
            {
                IsLoading = loading.IsLoading
            },

            _ => state
        };
    }

    private static ShoppingCart AddItemToCart(ShoppingCart cart, Product
product, int quantity)
    {
        var newItems = cart.Items.ToList();
        newItems.Add(new CartItem(product, quantity));

        return cart with
        {
            Items = newItems,
            Total = newItems.Sum(i => i.Product.Price * i.Quantity)
        };
    }
}

// Actions
record UserLoginAction(string Username, string Email);
record AddToCartAction(Product Product, int Quantity);
record SetLoadingAction(bool IsLoading);
```

Use Case 5: GraphQL/API Mutations

```
// Input models
record CreateUserInput(string Username, string Email, string Password);
record UpdateUserInput(int Id, string Email, string DisplayName);
record DeleteUserInput(int Id);

// Result models
record MutationResult<T>(bool Success, T Data, List<string> Errors);

record UserPayload(int Id, string Username, string Email, DateTime CreatedAt);

// Service
class UserService
{
    public MutationResult<UserPayload> CreateUser(CreateUserInput input)
    {
        // Validation
        var errors = new List<string>();

        if (string.IsNullOrWhiteSpace(input.Username))
            errors.Add("Username is required");
        if (errors.Any())
            return new MutationResult<UserPayload>(false, null, errors);

        // Create user
        var user = new UserPayload(
            1,
            input.Username,
            input.Email,
            DateTime.Now
        );

        return new MutationResult<UserPayload>(true, user, new List<string>
());
    }
}
```

Performance Considerations

Memory Allocation

```
// Record - heap allocated
record PersonRecord(string Name, int Age);

// Each instance allocates on heap
var p1 = new PersonRecord("Ali", 25);  // Heap allocation
var p2 = new PersonRecord("Sara", 30); // Heap allocation

// with expression - creates new object
var p3 = p1 with { Age = 26 };  // Another heap allocation
```

Struct vs Record Performance

```
// Benchmark example
record struct PointStruct(int X, int Y);
record PointRecord(int X, int Y);

// Struct - stack allocation, faster
void ProcessPoints()
{
    for (int i = 0; i < 1_000_000; i++)
    {
        var p = new PointStruct(i, i * 2);  // Stack, very fast
    }
}

// Record - heap allocation, slower
void ProcessRecords()
{
    for (int i = 0; i < 1_000_000; i++)
    {
        var p = new PointRecord(i, i * 2);  // Heap, slower, GC pressure
    }
}
```

Equality Comparison Cost

```
// Value equality compares ALL properties
record LargeRecord(
    string Prop1, string Prop2, string Prop3,
    int Prop4, int Prop5, int Prop6,
    DateTime Prop7, DateTime Prop8
);

var r1 = new LargeRecord(/* ... */);
var r2 = new LargeRecord(/* ... */);

// Must compare all 8 properties!
bool equal = r1 == r2;
```

Optimization Tips

Performance Tips

1. Use readonly record struct for small data

```
readonly record struct Point(int X, int Y);  // Fastest
```

2. Keep record structs small (<= 16 bytes)

```
// Good - 8 bytes (2 * int)
record struct Point2D(int X, int Y);

// Bad - 100+ bytes
record struct HugeStruct(/* many fields */);
```

3. Avoid with expressions in tight loops

```
// Bad - creates new object each iteration
var result = record;
for (int i = 0; i < 1000; i++)
{
    result = result with { Counter = i };  // Slow!
}

// Better - use mutable type in loop
```

4. Cache equality comparisons if used frequently

```
var isEqual = record1 == record2;  // Cache result if reused
```

Best Practices

1. Use Records for Data Models

```
// ✅ Good - immutable data model
record UserDto(int Id, string Name, string Email);

// ❌ Bad - mutable data model
record MutableUser(string Name)
{
    public string Name { get; set; } = Name;
}
```

2. Keep Records Immutable

```
// ✅ Good - immutable
record Address(string Street, string City, string Country);

// ❌ Bad - defeats purpose of records
record MutableAddress(string Street, string City)
{
    public string Street { get; set; } = Street;
    public string City { get; set; } = City;
}
```

3. Use Positional Syntax for Simple Records

```
// ✅ Good - concise, clear
record Point(int X, int Y);

// ❌ Verbose - unnecessary for simple data
record Point
{
    public int X { get; init; }
    public int Y { get; init; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

4. Validate in Properties, Not Constructor

```
// ✅ Good - validation in property
record Email(string Address)
{
    public string Address { get; init; } =
        IsValid(Address) ? Address : throw new ArgumentException("Invalid
email");

    private static bool IsValid(string email) => email.Contains("@");
}

// ❌ Harder to override/extend
record Email(string Address)
{
    public Email(string address) : this(address)
    {
        if (!address.Contains("@"))
            throw new ArgumentException("Invalid email");
    }
}
```

5. Use with Expressions for Updates

```
// ✅ Good - non-destructive mutation
var user = new User("Ali", 25);
```

```csharp
var older = user with { Age = 26 };

// ❌ Bad - making records mutable
var user = new MutableUser { Name = "Ali", Age = 25 };
user.Age = 26;  // Defeats immutability
```

6. Prefer Record Structs for Small Data

```csharp
// ✅ Good - small value type
readonly record struct Point(int X, int Y);

// ⚠️ Okay but less efficient for small data
record Point(int X, int Y);
```

7. Use Nominal Syntax for Complex Records

```csharp
// ✅ Good - many properties, clear structure
record Employee
{
    public int Id { get; init; }
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public string Email { get; init; }
    public string Phone { get; init; }
    public string Department { get; init; }
    public decimal Salary { get; init; }
    public DateTime HireDate { get; init; }
}

// ❌ Bad - too many positional parameters
record Employee(int Id, string FirstName, string LastName,
    string Email, string Phone, string Department,
    decimal Salary, DateTime HireDate);
```

8. Document Record Purpose

```csharp
/// <summary>
/// Represents an immutable user data transfer object for API responses.
/// </summary>
/// <param name="Id">Unique user identifier</param>
/// <param name="Username">User's username (must be unique)</param>
/// <param name="Email">User's email address</param>
record UserDto(int Id, string Username, string Email);
```

Summary

✓ **Key Takeaways**

Records are perfect for:

- ✅ DTOs (Data Transfer Objects)
- ✅ Value Objects
- ✅ Immutable data models
- ✅ API requests/responses
- ✅ Configuration objects
- ✅ Event sourcing
- ✅ State management

What records give you automatically:

- ✅ Value equality
- ✅ ToString with all properties
- ✅ Deconstruction
- ✅ with expressions
- ✅ Copy constructor
- ✅ GetHashCode
- ✅ Concise syntax (90% less code!)

Remember:

- Records are **reference types** by default
- Properties are **init-only** (immutable)
- Use **record struct** for small data
- **Value equality** compares all properties
- **with expressions** create copies with changes

By Abdullah Ali

Contact : +201012613453