C# Advanced Features - Deep Dive

Table of Contents

Implicit Type Local Variables (var)

Definition

The `var` keyword allows you to declare variables without explicitly specifying their type. The compiler **infers** the type from the initialization expression at **compile-time**.

Basic Syntax

```
var x = 5;           // Compiler infers: int
var name = "Ali";    // Compiler infers: string
var price = 99.99;   // Compiler infers: double
var isValid = true;  // Compiler infers: bool
```

How It Works

Compile-Time Type Inference

`var` is NOT dynamic typing - it's **static typing with type inference**:

```
var y = 10;
// y = "text"; // ❌ ERROR: Cannot convert string to int
```

Once the type is inferred, it **cannot change**!

Visual Representation

```
Source Code:          var x = 5;
                        ↓
Compiler Sees:        int x = 5;
                        ↓
Compiled IL:          int32 x = 5
```

Rules and Restrictions

⚠ Important Rules

1. **Must Initialize**: Cannot declare without initialization

```
var x;        // ❌ ERROR
var x = 5;    // ✅ OK
```

2. **Cannot Be null Without Cast**: Need explicit nullable type

```
var y = null;           // ❌ ERROR
var y = (int?)null;     // ✅ OK
int? z = null;          // ✅ OK (explicit type)
```

3. **Local Variables Only**: Cannot use for fields, properties, or parameters

```
class MyClass
{
    var field = 5;      // ❌ ERROR
    public var Prop { get; set; }  // ❌ ERROR

    void Method(var param) { }     // ❌ ERROR
}
```

4. **Must Be Compile-Time Constant**: Type must be determinable at compile time

```
var x = GetValue();  // ✅ OK if GetValue() return type is known
```

When to Use var

✓ Good Use Cases ✅

1. Complex Generic Types

```
// Without var (verbose)
Dictionary<string, List<int>> dict = new Dictionary<string, List<int>>();

// With var (cleaner)
var dict = new Dictionary<string, List<int>>();
```

2. Anonymous Types (Required!)

```
var person = new { Name = "Ali", Age = 25 };
// Cannot write explicit type for anonymous types
```

3. LINQ Queries

```
var results = from s in students
              where s.Age > 18
              select new { s.Name, s.Age };
```

4. Long Type Names

```
var connection = new SqlConnection(connectionString);
var reader = new StreamReader(filePath);
```

✕ Bad Use Cases ❌

1. Unclear Types

```
var data = GetData();  // What type is data?
```

2. Simple Types

```
var x = 5;         // Just use: int x = 5;
var name = "Ali"; // Just use: string name = "Ali";
```

3. When Explicit Type Adds Clarity

```
// Unclear
var result = Calculate();
```

```
// Clear
decimal result = Calculate();
```

var vs dynamic

Feature	var	dynamic
Type Resolution	Compile-time	Runtime
Type Checking	Static (compile-time)	Dynamic (runtime)
Performance	Fast	Slower (runtime overhead)
IntelliSense	Full support	Limited support
Type Changes	Cannot change	Can change
Usage	Prefer for known types	Use for interop/reflection

```
// var - Static typing
var x = 5;
// x = "text"; // ❌ Compile error

// dynamic - Dynamic typing
dynamic y = 5;
y = "text";    // ✅ OK - type changes at runtime
```

Common Patterns

Pattern 1: Object Initialization

```
var student = new Student
{
    Id = 1,
    Name = "Ali",
    Age = 20
};
```

Pattern 2: LINQ with var

```
var adults = students.Where(s => s.Age >= 18)
                     .OrderBy(s => s.Name)
                     .ToList();
```

Pattern 3: Tuple Deconstruction

```
var (id, name, age) = GetStudentInfo();
```

Pattern 4: Using Declarations

```
using var stream = new FileStream("file.txt", FileMode.Open);
```

Type Inference Examples

```
var a = 10;                    // int
var b = 10L;                   // long
var c = 10.5;                  // double
var d = 10.5f;                 // float
var e = 10.5m;                 // decimal
var f = "text";                // string
var g = new[] { 1, 2, 3 };     // int[]
var h = new List<string>();    // List<string>
```

Best Practices

Guidelines

1. ✅ Use `var` when type is obvious from right side
2. ✅ Use `var` for complex generic types
3. ✅ **Required** for anonymous types
4. ✅ Use in LINQ queries
5. ❌ Avoid when it reduces readability
6. ❌ Don't overuse for simple types
7. ✅ Be consistent in your codebase

Readonly vs Const

Overview

Both `readonly` and `const` create immutable values, but they work very differently and have distinct use cases.

Const Keyword

ⓘ Const Definition

`const` creates a **compile-time constant** that is embedded directly into the code during compilation.

Characteristics

```
class Employee
{
    public const int MaxAge = 120;
    public const string Company = "TechCorp";
    public const double Pi = 3.14159;
}
```

⚠ Const Rules

1. **Compile-Time Constant**: Value must be known at compile time
2. **Implicitly Static**: No need for `static` keyword
3. **Limited Types**: Only primitives and strings allowed
4. **Cannot Change**: Ever. Not even in constructor
5. **Embedded in Code**: Value copied to calling assemblies

What Can Be Const?

✅ Allowed Types:

- `bool`, `byte`, `short`, `int`, `long`
- `float`, `double`, `decimal`
- `char`, `string`
- `null` (for reference types)
- Enums

❌ Not Allowed:

- Classes (except string)
- Arrays
- Custom objects
- DateTime
- Anything requiring `new` keyword

Example

```
class MathConstants
{
    public const double Pi = 3.14159;
    public const int MaxValue = 100;
    public const string Version = "1.0.0";

    // ❌ ERRORS:
    // public const DateTime Now = DateTime.Now;      // Not compile-time
    // public const int[] Numbers = { 1, 2, 3 };      // No arrays
    // public const Employee Emp = new Employee();    // No objects
}
```

Readonly Keyword

> **ⓘ Readonly Definition**
>
> `readonly` creates a **runtime constant** that can be initialized in the constructor and never changed afterwards.

Characteristics

```
class Employee
{
    public readonly int Id;
    public readonly DateTime HireDate;
    public readonly string Department;

    public Employee(int id, string dept)
    {
        Id = id;                          // ✅ OK in constructor
        HireDate = DateTime.Now;          // ✅ OK - runtime value
        Department = dept;                // ✅ OK in constructor
```

```
    }
}
```

✓ **Readonly Features**

1. **Runtime Initialization**: Value determined at runtime
2. **Constructor Assignment**: Can be set in constructor
3. **Instance or Static**: Can be either
4. **Any Type**: Works with any data type
5. **Different Per Instance**: Each object can have different values

Two Initialization Options

Option 1: Inline Initialization

```
class Config
{
    public readonly int MaxConnections = 100;
    public readonly string AppName = "MyApp";
}
```

Option 2: Constructor Initialization

```
class Config
{
    public readonly int MaxConnections;
    public readonly Guid SessionId;

    public Config(int maxConn)
    {
        MaxConnections = maxConn;
        SessionId = Guid.NewGuid();  // Different for each instance
    }
}
```

Comprehensive Comparison

Feature	const	readonly
When Evaluated	Compile-time	Runtime
Static/Instance	Implicitly static	Can be either

Feature	const	readonly
Initialization	Declaration only	Declaration or constructor
Types Allowed	Primitives, strings	Any type
Value Per Instance	Same for all	Can differ per instance
Modification	Never	Never (after initialization)
Memory	No memory allocated (inline)	Memory allocated
Performance	Fastest (inlined)	Slightly slower
Versioning	Recompile needed	No recompile needed

Visual Representation

```
CONST - Compile Time:
Source Code:  const int X = 5;
              var result = X + 10;
                    ↓
Compiled IL:  var result = 5 + 10;  // Value embedded directly
              var result = 15;       // Even optimized!

READONLY - Runtime:
Source Code:  readonly int X = 5;
              var result = X + 10;
                    ↓
Compiled IL:  var result = obj.X + 10;  // Value read at runtime
```

Practical Examples

Example 1: Configuration Class

```
class AppConfig
{
    // Const for truly constant values
    public const string AppName = "MyApp";
    public const int Version = 1;

    // Readonly for runtime-determined values
    public readonly Guid InstanceId;
    public readonly DateTime StartTime;
    public readonly string ConnectionString;

    public AppConfig(string connString)
    {
```

```
        InstanceId = Guid.NewGuid();
        StartTime = DateTime.Now;
        ConnectionString = connString;
    }
}
```

Example 2: Mathematical Constants

```
class MathHelpers
{
    // Const for mathematical constants
    public const double Pi = 3.14159265359;
    public const double E = 2.71828182846;

    // Readonly for calculated values
    public static readonly double TwoPi = Pi * 2;
    public static readonly double SqrtTwo = Math.Sqrt(2);
}
```

Example 3: Array Constants

```
class DataConstants
{
    // ❌ Cannot do this with const
    // public const int[] Numbers = { 1, 2, 3 };

    // ✅ Use readonly for arrays
    public static readonly int[] Numbers = { 1, 2, 3 };
    public static readonly string[] Days = { "Mon", "Tue", "Wed" };
}
```

Versioning Implications

⚡ Critical Const Behavior

When you change a `const` value in a library:

Library v1.0:

```
public const int MaxSize = 100;
```

Client Code:

```
var size = Library.MaxSize;  // Compiled as: var size = 100;
```

Library v2.0:

```
public const int MaxSize = 200;  // Changed value
```

Result: Client code still has `100` embedded until recompiled! ⚠

With readonly: Client would automatically get new value (200) without recompilation.

When to Use What?

✓ **Use CONST when:**

- Value is truly constant (Pi, MaxValue)
- Value will **never** change
- Primitive types or strings
- Performance is critical
- Example: Mathematical constants, version numbers

✓ **Use READONLY when:**

- Value determined at runtime
- Different per instance
- Complex types needed
- Value might change in future versions
- Arrays or custom objects
- Example: Configuration, timestamps, GUIDs

Common Patterns

Pattern 1: Singleton with Readonly

```
class Singleton
{
    public static readonly Singleton Instance = new Singleton();
```

```
    private Singleton() { }
}
```

Pattern 2: Dependency Injection

```
class Service
{
    private readonly IRepository _repository;
    private readonly ILogger _logger;

    public Service(IRepository repo, ILogger logger)
    {
        _repository = repo;  // Readonly field initialized
        _logger = logger;
    }
}
```

Pattern 3: Immutable Objects

```
class Point
{
    public readonly int X;
    public readonly int Y;

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

Error Examples

```
class Examples
{
    public const int A = 100;
    public readonly int B;

    public Examples()
    {
        B = 50;           // ✅ OK
        // A = 200;       // ❌ ERROR: Cannot assign to const
    }
```

```
    public void Method()
    {
        // B = 100;       // ❌ ERROR: Cannot assign readonly outside
constructor
        // A = 300;       // ❌ ERROR: Cannot assign to const
    }
}
```

Division by Zero Example

```
class MathOperations
{
    public const int Zero = 0;

    public void Calculate()
    {
        // ⚠️ Compile-time error if const is zero
        Console.WriteLine(100 / Zero);  // Division by zero - compiler catches
this!
    }
}
```

The compiler can detect division by zero when using `const` because the value is known at compile time!

Primary Constructors (C# 12)

> **Definition**
>
> **Primary constructors** allow you to declare constructor parameters directly in the class declaration, eliminating the need for explicit constructor code and field declarations.

Traditional Constructor vs Primary Constructor

Before C# 12 (Traditional)

```
class Employee
{
    public int Id { get; set; }
```

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

    public Employee(int id, string name, int age)
    {
        Id = id;
        Name = name;
        Age = age;
    }
}
```

C# 12 (Primary Constructor)

```
class Employee(int id, string name, int age)
{
    public int Id { get; set; } = id;
    public string Name { get; set; } = name;
    public int Age { get; set; } = age;
}
```

✓ **Benefits**

- ✅ **Less boilerplate code**
- ✅ **More concise syntax**
- ✅ **Clearer intent**
- ✅ **Parameters available throughout class**

How It Works

ⓘ **Parameter Scope**

Primary constructor parameters are **in scope** for the entire class body:

```
class Employee(int id, string name, int age)
{
    // Parameters accessible in property initializers
    public int Id { get; set; } = id;
    public string Name { get; set; } = name;

    // Parameters accessible in methods
    public void Display()
    {
```

```
        Console.WriteLine($"{id}: {name}, Age: {age}");
    }

    // Parameters accessible in field initializers
    private string _info = $"{name} - {age} years";
}
```

Visual Representation

Traditional Way:

```
┌──────────────────────────┐
│ class Employee           │
│ {                        │
│   private int _id;       │ ← Field declaration
│   private string _name;  │
│                          │
│   public Employee(...)   │ ← Constructor
│   {                      │
│     _id = id;            │ ← Assignment
│     _name = name;        │
│   }                      │
│ }                        │
└──────────────────────────┘
```

Primary Constructor Way:

```
┌──────────────────────────┐
│ class Employee(int id,   │ ← Parameters in class declaration
│                string n) │
│ {                        │
│   // Parameters auto-    │
│   // available in scope  │
│ }                        │
└──────────────────────────┘
```

Detailed Example from Code

```
class Employee(int id = 0, string name = " ", int age = 6)
{
    public int Id { get; set; } = id;
    public string Name { get; set; } = name;
    public int Age { get; set; } = age;

    public override string ToString()
```

```
    {
        return $"{Id}-{Name}-{Age}";
    }
}

// Usage
var emp = new Employee(1, "Ali", 22);
Console.WriteLine(emp); // Output: 1-Ali-22
```

Features and Capabilities

1. Default Parameters

```
class Student(int id = 0, string name = "Unknown", int age = 18)
{
    public int Id { get; } = id;
    public string Name { get; } = name;
    public int Age { get; } = age;
}

// Usage
var s1 = new Student();                    // Uses all defaults
var s2 = new Student(1);                   // id=1, rest defaults
var s3 = new Student(1, "Ali");            // id=1, name="Ali", age=18
var s4 = new Student(1, "Ali", 20);        // All specified
```

2. Additional Constructors (Constructor Chaining)

```
class Employee(int id = 0, string name = " ", int age = 6)
{
    public int Id { get; set; } = id;
    public string Name { get; set; } = name;
    public int Age { get; set; } = age;

    // Additional constructor that chains to primary
    public Employee(int id, string name) : this(id, name, 5)
    {
        // Additional logic if needed
    }
}

// Usage
var emp1 = new Employee(1, "Ali", 25);  // Uses primary constructor
var emp2 = new Employee(2, "Sara");     // Uses secondary constructor (age=5)
```

3. Using Parameters in Expressions

```
class Product(string name, decimal price, int quantity)
{
    public string Name { get; } = name.ToUpper();  // Transform parameter
    public decimal Price { get; } = price;
    public int Quantity { get; } = quantity;

    // Computed property using parameters
    public decimal TotalValue => price * quantity;

    // Method using parameters
    public string GetInfo() => $"{name}: {quantity} items at ${price}";
}
```

4. Validation in Primary Constructors

```
class BankAccount(string accountNumber, decimal initialBalance)
{
    public string AccountNumber { get; } =
!string.IsNullOrWhiteSpace(accountNumber)
        ? accountNumber
        : throw new ArgumentException("Account number required");

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

Parameters vs Properties

⚠ Important Distinction

Primary constructor **parameters** are NOT automatically properties:

```
class Employee(int id, string name)
{
    // Parameters are just in scope, not stored

    // To store them, create properties:
    public int Id { get; } = id;
```

```
    public string Name { get; } = name;
}
```

Capturing Parameters

Parameter Lifetime

Parameters are **captured** and available throughout the class:

```
class Logger(string logPath)
{
    // logPath parameter captured and used in methods
    public void Log(string message)
    {
        File.AppendAllText(logPath, $"{DateTime.Now}: {message}\n");
    }

    public void LogError(string error)
    {
        File.AppendAllText(logPath, $"ERROR: {error}\n");
    }
}
```

Common Patterns

Pattern 1: Immutable Objects

```
class Point(int x, int y)
{
    public int X { get; } = x;  // Read-only property
    public int Y { get; } = y;  // Read-only property

    public double DistanceFromOrigin() => Math.Sqrt(x * x + y * y);
}
```

Pattern 2: Dependency Injection

```
class OrderService(IRepository repository, ILogger logger)
{
    private readonly IRepository _repository = repository;
    private readonly ILogger _logger = logger;
```

```
    public void ProcessOrder(Order order)
    {
        _logger.Log("Processing order");
        _repository.Save(order);
    }
}
```

Pattern 3: Configuration Classes

```
class AppSettings(string apiKey, string connectionString, int timeout = 30)
{
    public string ApiKey { get; } = apiKey;
    public string ConnectionString { get; } = connectionString;
    public int TimeoutSeconds { get; } = timeout;

    public bool IsValid() => !string.IsNullOrEmpty(apiKey) &&
                             !string.IsNullOrEmpty(connectionString);
}
```

Combining with Records

```
record Employee(int Id, string Name, int Age)
{
    // Records with primary constructors are even more concise
    // Properties created automatically!

    public void Display() => Console.WriteLine($"{Id}: {Name}");
}
```

Best Practices

✓ Guidelines

1. ✅ Use for simple classes with straightforward initialization
2. ✅ Combine with default parameters for flexibility
3. ✅ Use for dependency injection scenarios
4. ✅ Keep validation simple or extract to properties
5. ❌ Avoid complex logic in parameter expressions
6. ❌ Don't use for classes with complex initialization
7. ✅ Consider using with `required` keyword for mandatory properties

Limitations

> ⚠ **Important Limitations**
>
> 1. Cannot have multiple primary constructors (only one per class)
> 2. Parameters don't automatically become properties (must explicitly create)
> 3. Cannot access primary constructor parameters from nested types
> 4. Struct primary constructors must explicitly assign all fields

---

Auto-Property Initializers

> 📋 **Definition**
>
> **Auto-property initializers** allow you to initialize properties directly in their declaration, without needing a constructor.

Basic Syntax

```
class Subject
{
    public int Id { get; set; } = 0;            // Initialized to 0
    public string Name { get; set; } = "Unknown"; // Initialized to "Unknown"
    public int Duration { get; set; } = 6;      // Initialized to 6
}
```

Before vs After

Before C# 6 (Without Auto-Property Initializers)

```
class Subject
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Duration { get; set; }

    // Had to initialize in constructor
    public Subject()
    {
```

```
        Id = 0;
        Name = "Unknown";
        Duration = 6;
    }
}
```

After C# 6 (With Auto-Property Initializers)

```
class Subject
{
    public int Id { get; set; } = 0;
    public string Name { get; set; } = "Unknown";
    public int Duration { get; set; } = 6;

    // No constructor needed for default values!
}
```

✓ **Benefits**

✅ **Cleaner code** - No constructor clutter
✅ **Default values visible** at property declaration
✅ **Combine with constructors** for flexibility
✅ **Works with read-only properties**
✅ **Compile-time constants** or expressions

Features and Capabilities

1. Different Value Types

```
class Configuration
{
    // Primitives
    public int MaxConnections { get; set; } = 100;
    public bool IsEnabled { get; set; } = true;
    public double Timeout { get; set; } = 30.5;

    // Strings
    public string AppName { get; set; } = "MyApp";

    // Objects
    public DateTime CreatedAt { get; set; } = DateTime.Now;
    public List<string> Tags { get; set; } = new List<string>();
```

```
    public Guid Id { get; set; } = Guid.NewGuid();
}
```

2. Read-Only Properties with Initializers

```
class ImmutableConfig
{
    public int MaxSize { get; } = 1000;              // Cannot be changed
after init
    public string Version { get; } = "1.0.0";
    public DateTime Timestamp { get; } = DateTime.UtcNow;
}

// Usage
var config = new ImmutableConfig();
// config.MaxSize = 500;  // ❌ ERROR: Property is read-only
```

3. Combining with Constructors

```
class Subject
{
    public int Id { get; set; } = 0;            // Default value
    public string Name { get; set; } = "Unknown";
    public int Duration { get; set; } = 6;

    // Constructor can override defaults
    public Subject(int id, string name, int duration)
    {
        Id = id;           // Overrides default
        Name = name;       // Overrides default
        Duration = duration; // Overrides default
    }
}

// Usage
var s1 = new Subject();                      // Uses initializer defaults
var s2 = new Subject(1, "C#", 40);          // Uses constructor values
```

4. Complex Expressions

```
class Advanced
{
    // Computed values
    public string Username { get; set; } = Environment.UserName;
```

```
    public string MachineName { get; set; } = Environment.MachineName;

    // Collection initializers
    public List<int> Numbers { get; set; } = new List<int> { 1, 2, 3, 4, 5 };
    public Dictionary<string, int> Ages { get; set; } = new Dictionary<string,
int>
    {
        { "Alice", 25 },
        { "Bob", 30 }
    };

    // Complex objects
    public Person Owner { get; set; } = new Person { Name = "Admin", Role =
"Owner" };
}
```

Order of Execution

> **Initialization Order**
>
> When an object is created, this is the execution order:
>
> ```
> 1. Auto-property initializers (run first)
>    ↓
> 2. Field initializers
>    ↓
> 3. Base class constructor
>    ↓
> 4. This class constructor
> ```

Example:

```
class Demo
{
    public int A { get; set; } = 1;     // Runs FIRST
    public int B { get; set; } = 2;

    public Demo()
    {
        A = 10;  // Runs LAST, overrides initializer value
        Console.WriteLine($"A={A}, B={B}");  // Output: A=10, B=2
```

```
    }
}
```

Practical Examples

Example 1: Application Settings

```
class AppSettings
{
    public string AppName { get; set; } = "My Application";
    public int MaxRetries { get; set; } = 3;
    public bool EnableLogging { get; set; } = true;
    public string LogPath { get; set; } = @"C:\Logs\app.log";
    public TimeSpan Timeout { get; set; } = TimeSpan.FromMinutes(5);
}
```

Example 2: Entity with Defaults

```
class Order
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public string Status { get; set; } = "Pending";
    public decimal Total { get; set; } = 0m;
    public List<OrderItem> Items { get; set; } = new List<OrderItem>();
}
```

Example 3: Game Character

```
class Character
{
    public string Name { get; set; } = "Hero";
    public int Health { get; set; } = 100;
    public int Mana { get; set; } = 50;
    public int Level { get; set; } = 1;
    public List<string> Inventory { get; set; } = new List<string> { "Sword",
"Shield" };
}
```

Combining with Primary Constructors

```
class Subject(int id = 0, string name = " ", int duration = 6)
{
```

```
    // Primary constructor parameters used as initializers
    public int Id { get; set; } = id;
    public string Name { get; set; } = name;
    public int Duration { get; set; } = duration;
}
```

With Object Initializers

```
class Student
{
    public int Id { get; set; } = 0;
    public string Name { get; set; } = "Unknown";
    public int Age { get; set; } = 18;
}

// Object initializer syntax
var student = new Student
{
    Id = 1,          // Overrides default
    Name = "Ali",    // Overrides default
    Age = 20         // Overrides default
};

var student2 = new Student
{
    Name = "Sara"    // Only override name, others use defaults
};
```

Static Properties

Static Auto-Property Initializers

Works with static properties too:

```
class AppConstants
{
    public static string Version { get; } = "1.0.0";
    public static int MaxUsers { get; } = 1000;
    public static DateTime StartupTime { get; } = DateTime.Now;
}
```

Best Practices

✓ **Guidelines**

1. ✅ Use for sensible default values
2. ✅ Initialize collections to prevent null checks
3. ✅ Use with read-only properties for immutability
4. ✅ Prefer over constructor initialization for simple defaults
5. ✅ Combine with constructors for flexibility
6. ❌ Avoid expensive operations in initializers
7. ❌ Don't initialize with dependencies (use DI instead)

Common Patterns

Pattern 1: Non-Null Collections

```
class Blog
{
    public List<Post> Posts { get; set; } = new List<Post>();
    public List<Comment> Comments { get; set; } = new List<Comment>();

    // No need to check for null:
    public void AddPost(Post post)
    {
        Posts.Add(post);  // Safe - never null
    }
}
```

Pattern 2: Sensible Defaults

```
class User
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public bool IsActive { get; set; } = true;
    public string Role { get; set; } = "User";
}
```

Deconstructors

Definition

A **deconstructor** is a method that allows you to "break apart" an object into its individual components using tuple syntax. It's the opposite of a constructor!

Basic Concept

```
Constructor:  Parts  →  Object
             (a, b, c) → new MyClass(a, b, c)

Deconstructor: Object  →  Parts
               myObject  → (a, b, c)
```

Syntax and Implementation

```
class Subject
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Duration { get; set; }

    // Deconstructor method
    public void Deconstruct(out int id, out string name, out int duration)
    {
        id = this.Id;
        name = this.Name;
        duration = this.Duration;
    }
}
```

Method Signature Requirements

- Name must be exactly `Deconstruct`
- Must be public or internal
- Return type must be `void`
- Uses `out` parameters
- Can have multiple overloads with different parameter counts

Using Deconstructors

Method 1: Explicit Out Variables

```
Subject s = new Subject(1, "C#", 60);

// Traditional way
s.Deconstruct(out int id, out string name, out int duration);
Console.WriteLine(name);  // Output: C#
```

Method 2: Tuple Deconstruction (Recommended)

```
Subject s = new Subject(1, "C#", 60);

// Clean tuple deconstruction syntax
var (id, name, duration) = s;

Console.WriteLine(name);      // Output: C#
Console.WriteLine(duration);  // Output: 60
```

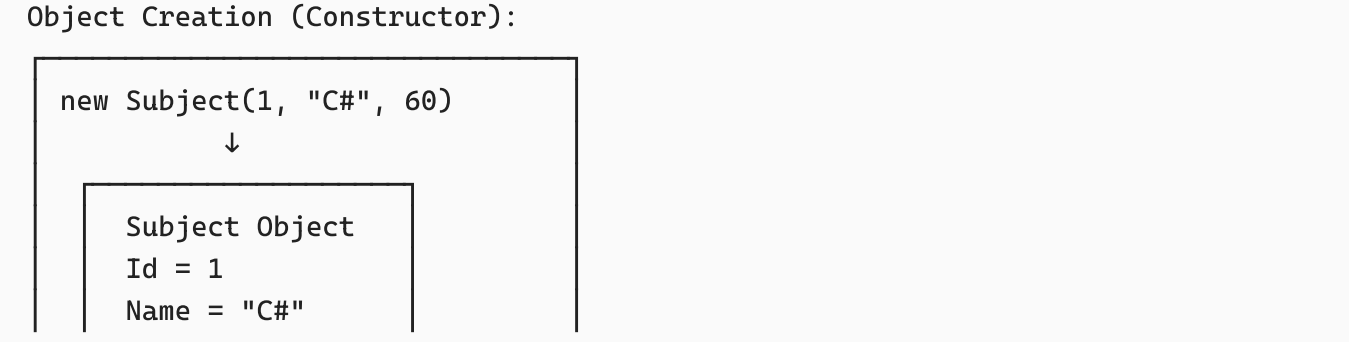
Method 3: With Type Inference

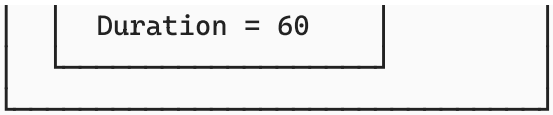
```
Subject s = new Subject(1, "C#", 60);

// Compiler infers types
var (id, name, duration) = s;
```

Method 4: With Explicit Types

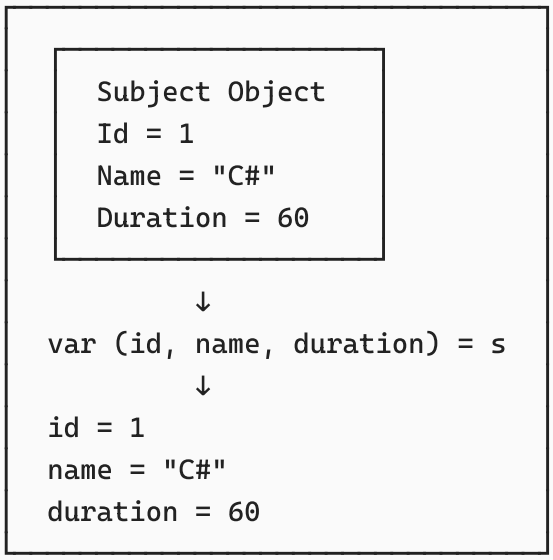
```
Subject s = new Subject(1, "C#", 60);

// Explicit types
(int id, string name, int duration) = s;
```

Visual Representation





Object Deconstruction (Deconstructor):



Discarding Values

Underscore Discard Pattern

Use `_` to ignore values you don't need:

```
var (_, name, _) = subject;  // Only care about name
Console.WriteLine(name);
```

Multiple Deconstructors (Overloading)

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }

    // Deconstruct into 2 values
    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
```

```
    // Deconstruct into 3 values
    public void Deconstruct(out string firstName, out string lastName, out int
age)
    {
        firstName = FirstName;
        lastName = LastName;
        age = Age;
    }
}

// Usage
var person = new Person { FirstName = "John", LastName = "Doe", Age = 30 };

var (first, last) = person;           // Uses 2-parameter version
var (first, last, age) = person;      // Uses 3-parameter version
```

Practical Examples

Example 1: Point Class

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public void Deconstruct(out int x, out int y)
    {
        x = X;
        y = Y;
    }
}

// Usage
var point = new Point { X = 10, Y = 20 };
var (x, y) = point;
Console.WriteLine($"X: {x}, Y: {y}");  // X: 10, Y: 20
```

Example 2: Student Record

```
class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
```

```
    public int Age { get; set; }
    public string Department { get; set; }

    public void Deconstruct(out int id, out string name)
    {
        id = Id;
        name = Name;
    }

    public void Deconstruct(out int id, out string name, out int age, out
string dept)
    {
        id = Id;
        name = Name;
        age = Age;
        dept = Department;
    }
}

// Usage - Choose what you need
var student = new Student { Id = 1, Name = "Ali", Age = 20, Department = "CS"
};

var (id, name) = student;                    // Basic info
var (id, name, age, dept) = student;         // Full info
var (_, name, _, _) = student;               // Only name
```

Example 3: DateTime Extension

```
static class DateTimeExtensions
{
    public static void Deconstruct(this DateTime dt, out int year, out int
month, out int day)
    {
        year = dt.Year;
        month = dt.Month;
        day = dt.Day;
    }
}

// Usage
var today = DateTime.Now;
var (year, month, day) = today;
Console.WriteLine($"{year}-{month:D2}-{day:D2}");
```

Common Use Cases

When to Use Deconstructors

1. Extracting Coordinates

```
var (x, y, z) = GetPosition();
```

2. Processing Return Values

```
var (success, message, data) = ProcessRequest();
if (success)
{
    Console.WriteLine(data);
}
```

3. Working with Key-Value Pairs

```
foreach (var (key, value) in dictionary)
{
    Console.WriteLine($"{key}: {value}");
}
```

4. Tuple-like Behavior for Custom Classes

```
var (status, result) = operation.Execute();
```

Deconstruction in Foreach

```
var students = new List<Student>
{
    new Student { Id = 1, Name = "Ali" },
    new Student { Id = 2, Name = "Sara" },
    new Student { Id = 3, Name = "Omar" }
};

// Deconstruct each student in the loop
foreach (var (id, name) in students)
{
```

```
    Console.WriteLine($"{id}: {name}");
}
```

With Pattern Matching

```
Subject subject = GetSubject();

var result = subject switch
{
    (1, "C#", _) => "Basic C# course",
    (_, _, > 100) => "Long course",
    var (_, name, _) => $"Course: {name}"
};
```

Best Practices

✓ Guidelines

1. ✅ Implement for classes that represent data/tuples
2. ✅ Provide multiple overloads for flexibility
3. ✅ Keep deconstructor logic simple
4. ✅ Use meaningful parameter names
5. ✅ Consider what consumers need most
6. ❌ Don't include expensive computations
7. ❌ Avoid side effects

Comparison with Tuples

```
// Tuple (anonymous)
var tuple = (Id: 1, Name: "Ali", Age: 20);
var (id, name, age) = tuple;

// Custom class with deconstructor
var student = new Student { Id = 1, Name = "Ali", Age = 20 };
var (id, name, age) = student;
```

Both use same deconstruction syntax, but custom classes offer:

- Named types
- Methods and behavior

- Validation
- Better semantics

By Abdullah Ali

Contact : +201012613453