C# Structs vs Classes, Properties & Object Initialization - Complete Guide

1. Properties in C#

Properties provide **controlled access** to private fields while maintaining encapsulation.

1.1 Full Property Implementation

Example from code:

```
int real;  // Private backing field

public int Real
{
    set
    {
        if (value > 0)
            real = value;
        else
            throw new Exception("invalid real value");
    }
    get
    {
        return real;
    }
}
```

Components:

- **Backing field** (`real`): Stores the actual data
- **Getter** (`get`): Returns the value
- **Setter** (`set`): Sets the value with validation
- `value` keyword: Represents the incoming value in setter

How it works:

```
ComplexNum c = new ComplexNum();
c.Real = 5;   // Calls setter: set { if (5 > 0) real = 5; }
c.Real = -3;  // Calls setter: throws Exception (validation fails)
int x = c.Real;  // Calls getter: return real;
```

1.2 Auto-Implemented Properties

Example from code:

```
public int Img { set; get; }  // Compiler creates hidden backing field
public int Real { set; get; }
```

What the compiler generates:

```
// Behind the scenes:
private int <Img>k__BackingField;
public int Img
{
    get { return <Img>k__BackingField; }
    set { <Img>k__BackingField = value; }
}
```

Advantages:

- ✅ Concise syntax
- ✅ Less boilerplate code
- ✅ Easy to add validation later
- ✅ Maintains encapsulation

Limitations:

- ❌ Cannot add custom logic without refactoring
- ❌ Cannot access backing field directly

1.3 Property Access Modifiers

```
// Read-only property (public get, no set)
public int ReadOnly { get; }

// Write-only property (rare, public set, no get)
public int WriteOnly { set; }

// Public get, private set
public int Id { get; private set; }

// Public property with private backing field
private int age;
public int Age
```

```
{
    get { return age; }
    private set { age = value; }
}
```

1.4 Init-Only Properties (C# 9.0+)

Example from code (commented):

```
public required int Img { init; get; }
public int Real { init; get; }
```

Characteristics:

- Can only be set during **object initialization**
- Immutable after construction
- `required` keyword makes the property mandatory

Usage:

```
// ✅ Valid - set during initialization
ComplexNum c = new ComplexNum { Img = 3, Real = 5 };

// ❌ Invalid - cannot modify after initialization
c.Img = 10;  // Compile error!
```

Benefits:

- Creates **immutable objects**
- Prevents accidental modification
- Better for thread safety
- Functional programming style

1.5 Property Patterns Comparison

```
// 1. Full property with validation
private int real;
public int Real
{
    set { if (value > 0) real = value; }
    get { return real; }
}
```

```csharp
// 2. Auto-implemented property
public int Real { get; set; }

// 3. Init-only property
public int Real { get; init; }

// 4. Required init property (C# 11)
public required int Real { get; init; }

// 5. Expression-bodied property (C# 7.0+)
private int real;
public int Real
{
    get => real;
    set => real = value > 0 ? value : throw new Exception("invalid");
}

// 6. Computed property (read-only)
public string Display => $"{Real}+{Img}i";
```

---

2. Object Initialization Syntax

2.1 Object Initializer Syntax

Example from code:

```csharp
ComplexNum c = new ComplexNum() { Real = 3, Img = 4 };
```

Behind the scenes:

```csharp
// Compiler translates to:
ComplexNum c = new ComplexNum();  // Call constructor
c.Real = 3;                        // Set property
c.Img = 4;                         // Set property
```

Advantages:

- ✅ Cleaner, more readable code
- ✅ Can initialize specific properties only
- ✅ Works with parameterless constructor
- ✅ Better for complex objects

2.2 Different Initialization Styles

Example from code:

```
// 1. Empty initializer (uses default constructor)
ComplexNum c = new ComplexNum() { };

// 2. Partial initialization (only Img)
ComplexNum c = new ComplexNum() { Img = 3 };
// Real uses default value from constructor (1)

// 3. Full initialization
ComplexNum c = new ComplexNum() { Real = 3, Img = 4 };

// 4. Traditional way
ComplexNum c = new ComplexNum();
c.Real = 4;
c.Img = 3;
```

2.3 Target-Typed New Expressions (C# 9.0+)

Example from code:

```
ComplexNum c1 = new();  // Type inferred from left side
```

More examples:

```
// Old way
ComplexNum c1 = new ComplexNum();
ComplexNum c2 = new ComplexNum(2, 3);

// New way (shorter)
ComplexNum c1 = new();
ComplexNum c2 = new(2, 3);

// Especially useful with long type names
Dictionary<string, List<int>> dict = new();
```

2.4 Collection Initializers

```
// Array initialization
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
int[] numbers = { 1, 2, 3, 4, 5 };  // Shorter
```

```
// List initialization
List<int> list = new List<int> { 1, 2, 3, 4, 5 };
List<int> list = new() { 1, 2, 3, 4, 5 };  // C# 9.0+

// Dictionary initialization
var dict = new Dictionary<string, int>
{
    { "one", 1 },
    { "two", 2 },
    ["three"] = 3  // Index initializer (C# 6.0+)
};

// Complex object with collection
class Student
{
    public string Name { get; set; }
    public List<string> Subjects { get; set; }
}

Student s = new Student
{
    Name = "Ali",
    Subjects = new List<string> { "C#", "SQL", "HTML" }
};

// Nested initialization
Student s = new()
{
    Name = "Ali",
    Subjects = new() { "C#", "SQL", "HTML" }
};
```

---

3. Constructors

3.1 Parameterless Constructor

Example from code:

```
public ComplexNum()
{
    Real = 1;
```

```
    Img = 1;
}
```

Usage:

```
ComplexNum c = new ComplexNum();
Console.WriteLine(c.getstring());  // Output: 1+1i
```

Important note:

- If you define **any constructor**, the compiler **does NOT** generate a default parameterless constructor
- You must explicitly define it if needed

3.2 Parameterized Constructors

Example from code:

```
public ComplexNum(int _real, int _img)
{
    Real = _real;
    Img = _img;
}

public ComplexNum(int _real)
{
    Real = _real;
    // Img uses default value (0 for int)
}
```

Constructor overloading:

```
ComplexNum c1 = new ComplexNum();           // 1+1i (uses parameterless)
ComplexNum c2 = new ComplexNum(3, 4);       // 3+4i (uses two-parameter)
ComplexNum c3 = new ComplexNum(5);          // 5+0i (uses one-parameter)
```

3.3 Constructor Execution Order

```
class ComplexNum
{
    public int Real { get; set; } = 10;  // 1. Field initializers
    public int Img { get; set; }
```

```
    public ComplexNum()                    // 2. Constructor body
    {
        Real = 1;
        Img = 1;
    }
}

ComplexNum c = new ComplexNum();
// Real = 1 (constructor overrides field initializer)
// Img = 1 (set by constructor)
```

Order of execution:

1. Field initializers (property default values)
2. Base class constructor (if inherited)
3. Current class constructor body

---

4. Struct vs Class - Comprehensive Comparison

4.1 Basic Differences

```
// Struct - Value Type
struct ComplexNum
{
    public int Real { get; set; }
    public int Img { get; set; }
}

// Class - Reference Type
class ComplexNum
{
    public int Real { get; set; }
    public int Img { get; set; }
}
```

4.2 Memory Allocation

```
STRUCT (Value Type):
┌──────────────────────┐
│  Stack Memory        │
├──────────────────────┤
│  c1:                 │
```

```
│  ├─ Real: 3         │  ← Stored directly on stack
│  └─ Img: 4          │
│                      │
│  c2:                 │
│  ├─ Real: 3         │  ← Separate copy
│  └─ Img: 4          │
└──────────────────────┘

CLASS (Reference Type):
┌──────────────────────┐     ┌───────────────────┐
│  Stack Memory        │     │  Heap Memory      │
├──────────────────────┤     ├───────────────────┤
│  c1: ────────────────┼────→│  Object:          │
│                      │     │  ├─ Real: 3       │
│  c2: ────────────────┼──┐  │  └─ Img: 4        │
└──────────────────────┘  │  └───────────────────┘
                          │
                          └───→ (points to same object)
```

4.3 Assignment Behavior

Struct (copy by value):

```
struct ComplexNum { public int Real; public int Img; }

ComplexNum c1 = new ComplexNum { Real = 3, Img = 4 };
ComplexNum c2 = c1;  // COPIES all values

c2.Real = 10;
Console.WriteLine(c1.Real);  // 3 (unchanged)
Console.WriteLine(c2.Real);  // 10
```

Class (copy by reference):

```
class ComplexNum { public int Real; public int Img; }

ComplexNum c1 = new ComplexNum { Real = 3, Img = 4 };
ComplexNum c2 = c1;  // COPIES the reference (both point to same object)

c2.Real = 10;
Console.WriteLine(c1.Real);  // 10 (changed!)
Console.WriteLine(c2.Real);  // 10 (same object)
```

4.4 Struct Restrictions

Struct limitations:

```
struct ComplexNum
{
    // ❌ Cannot have parameterless constructor in C# < 10
    // public ComplexNum() { }  // Compile error (before C# 10)

    // ✅ Allowed in C# 10+
    public ComplexNum()
    {
        Real = 1;
        Img = 1;
    }

    // ❌ Cannot initialize fields directly (before C# 11)
    // public int Real = 0;  // Compile error

    // ✅ Can initialize properties (C# 10+)
    public int Real { get; set; } = 0;

    // ❌ Cannot inherit from classes
    // struct ComplexNum : SomeClass  // Compile error

    // ✅ Can implement interfaces
    public ComplexNum : IComparable<ComplexNum> { }
}
```

4.5 When to Use Struct vs Class

Criteria	Use Struct	Use Class
Size	Small (< 16 bytes)	Any size
Mutability	Immutable preferred	Can be mutable
Lifetime	Short-lived	Long-lived
Identity	Value equality	Reference equality
Inheritance	No inheritance needed	Need inheritance
Performance	Avoid heap allocation	Standard object model
Example	Point, Color, DateTime	Person, Order, Account

Good struct candidates:

```
struct Point
{
    public int X { get; init; }
    public int Y { get; init; }
}

struct Color
{
    public byte R { get; init; }
    public byte G { get; init; }
    public byte B { get; init; }
}

struct Money
{
    public decimal Amount { get; init; }
    public string Currency { get; init; }
}
```

Should be classes:

```
class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Order> Orders { get; set; }
    // Large, mutable, complex relationships
}

class BankAccount
{
    public string AccountNumber { get; set; }
    public decimal Balance { get; set; }
    // Needs reference semantics
}
```

4.6 Performance Considerations

Stack vs Heap allocation:

```
// Struct - Stack allocated (faster)
void ProcessPoints()
{
    Point p1 = new Point(1, 2);  // Stack
```

```
    Point p2 = new Point(3, 4);  // Stack
    // No garbage collection needed
}

// Class - Heap allocated
void ProcessComplexNumbers()
{
    ComplexNum c1 = new ComplexNum(1, 2);  // Heap
    ComplexNum c2 = new ComplexNum(3, 4);  // Heap
    // Garbage collector will clean up later
}
```

Boxing/Unboxing (structs only):

```
int x = 5;                  // Struct (value type)
object obj = x;             // Boxing (copy to heap)
int y = (int)obj;           // Unboxing (copy back to stack)

// Boxing is expensive! Avoid in loops
List<object> list = new List<object>();
for (int i = 0; i < 1000; i++)
{
    list.Add(i);  // ❌ Boxes each int (1000 heap allocations!)
}
```

---

5. Destructors (Finalizers)

Example from code:

```
~ComplexNum()
{
    Console.WriteLine("destructor");
}
```

5.1 What is a Destructor?

- Called **finalizer** in C# (destructor is C++ terminology)
- Invoked by **garbage collector** before object is destroyed
- **Cannot be called directly**
- **Only for classes**, not structs
- Used for **unmanaged resource cleanup**

5.2 Destructor Syntax

```
class ComplexNum
{
    // Constructor
    public ComplexNum()
    {
        Console.WriteLine("Constructor called");
    }

    // Destructor (finalizer)
    ~ComplexNum()
    {
        Console.WriteLine("Destructor called");
    }
}
```

Characteristics:

- Same name as class with `~` prefix
- No parameters
- No access modifier
- No return type
- Cannot be overloaded

5.3 When Destructors Run

```
void TestDestructor()
{
    ComplexNum c = new ComplexNum();
    Console.WriteLine("Object created");
    // c goes out of scope here
}  // Object eligible for GC, but destructor runs later!

TestDestructor();
Console.WriteLine("Method finished");
GC.Collect();  // Force garbage collection
GC.WaitForPendingFinalizers();  // Wait for destructors

// Output:
// Constructor called
// Object created
```

```
// Method finished
// Destructor called  ← Runs later, non-deterministically
```

5.4 Problems with Destructors

Non-deterministic execution:

```
class FileHandler
{
    private FileStream file;

    public FileHandler(string path)
    {
        file = new FileStream(path, FileMode.Open);
    }

    ~FileHandler()
    {
        file.Close();  // ❌ Problem: May run much later!
    }
}

// File might stay open for a long time!
```

5.5 Better Alternative: IDisposable

```
class FileHandler : IDisposable
{
    private FileStream file;

    public FileHandler(string path)
    {
        file = new FileStream(path, FileMode.Open);
    }

    // Deterministic cleanup
    public void Dispose()
    {
        if (file != null)
        {
            file.Close();
            file.Dispose();
            file = null;
        }
    }
```

```
    // Finalizer as safety net
    ~FileHandler()
    {
        Dispose();
    }
}

// Usage with 'using' statement (automatic disposal)
using (FileHandler handler = new FileHandler("file.txt"))
{
    // Use handler
}  // Dispose() called automatically here!

// Or with C# 8.0+ using declaration:
using FileHandler handler = new FileHandler("file.txt");
// Dispose() called at end of scope
```

5.6 Destructor Best Practices

✅ **DO:**

- Implement `IDisposable` instead of relying on finalizers
- Use finalizers only as a safety net
- Keep finalizer code simple and fast
- Call `GC.SuppressFinalize(this)` in Dispose

❌ **DON'T:**

- Rely on finalizers for critical cleanup
- Perform complex operations in finalizers
- Access managed objects in finalizers (may be collected)
- Use finalizers for structs (not allowed)

Proper pattern:

```
class ResourceManager : IDisposable
{
    private bool disposed = false;
    private IntPtr unmanagedResource;

    public void Dispose()
    {
        Dispose(true);
```

```
        GC.SuppressFinalize(this);  // Tell GC not to call finalizer
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose managed resources
            }

            // Free unmanaged resources
            if (unmanagedResource != IntPtr.Zero)
            {
                // Release unmanaged resource
                unmanagedResource = IntPtr.Zero;
            }

            disposed = true;
        }
    }

    ~ResourceManager()
    {
        Dispose(false);  // Only cleanup unmanaged resources
    }
}
```

---

6. Method Signature vs Method Header

From code comment:

```
// header: public string getstring(int x)
// signature: string(int)
```

Method Header (Declaration):

The **complete method declaration** including access modifier, return type, name, and parameters.

```
public string getstring(int x)
```

Method Signature:

The **unique identifier** of a method, consisting of:

- Method name
- Parameter types (number, type, and order)
- **NOT** including return type or parameter names

```
getstring(int)
```

Why Signatures Matter:

```
class Example
{
    // Different signatures - valid overloading
    public void Print(int x) { }           // Signature: Print(int)
    public void Print(string x) { }        // Signature: Print(string)
    public void Print(int x, int y) { }    // Signature: Print(int, int)

    // ❌ Same signature - compile error!
    // public int Print(int x) { }         // Signature: Print(int) -
duplicate!

    // ❌ Same signature - compile error!
    // public void Print(int y) { }        // Signature: Print(int) -
parameter name doesn't matter!
}
```

Key Takeaways Summary

1. **Full Properties**: Backing field + getter/setter with validation logic
2. **Auto-Implemented Properties**: Compiler generates backing field automatically
3. **Init-Only Properties**: Can only be set during initialization (immutable)
4. **Required Properties**: Must be initialized when creating object (C# 11)
5. **Object Initializer**: Clean syntax for setting properties at creation
6. **Target-Typed New**: Shorter `new()` syntax (C# 9.0+)
7. **Constructors**: Initialize object state, can be overloaded
8. **Struct vs Class**: Value type vs reference type, stack vs heap
9. **Struct Usage**: Small, immutable, value-semantic types

10. **Class Usage**: Complex, mutable, reference-semantic objects
11. **Destructors**: Non-deterministic cleanup, prefer IDisposable
12. **Method Signature**: Name + parameter types (for overloading resolution)

Understanding these fundamentals is critical for **proper object-oriented design**, **memory management**, and building **efficient C# applications**!

By Abdullah Ali

Contact : +201012613453