Complete Guide to Constructors in C#

Table of Contents

What is a Constructor?

Definition

A **constructor** is a special method that is automatically called when an object is created. It initializes the object's state by setting initial values to fields and properties.

Key Characteristics

Constructor Features

1. **Same Name**: Constructor name = Class name (exact match)
2. **No Return Type**: Not even `void`
3. **Automatic Invocation**: Called when object is created with `new`
4. **Initialization Purpose**: Set up initial state

5. **Can Be Overloaded**: Multiple constructors with different parameters
6. **Access Modifiers**: Can be public, private, protected, internal

Basic Syntax

```
class ClassName
{
    // Constructor
    public ClassName()
    {
        // Initialization code
    }
}
```

Purpose of Constructors

✓ **Why Use Constructors?**

- ✅ Initialize object state
- ✅ Allocate resources
- ✅ Validate parameters
- ✅ Setup dependencies
- ✅ Configure initial behavior
- ✅ Enforce object creation rules

Visual Representation

```
Object Creation Process:
┌─────────────────────────────────────┐
│ var obj = new MyClass(params);      │
│              ↓                      │
│  1. Memory allocated for object     │
│              ↓                      │
│  2. Fields initialized to defaults  │
│              ↓                      │
│  3. Constructor executed            │
│              ↓                      │
│  4. Object reference returned       │
└─────────────────────────────────────┘
```

Default Constructor

Definition

A **default constructor** is a parameterless constructor. If you don't define ANY constructor, C# automatically provides one.

Compiler-Generated Default Constructor

```
class Student
{
    public int Id;
    public string Name;

    // No constructor defined
    // Compiler automatically generates:
    // public Student() { }
}

// Usage
var student = new Student(); // Works! Uses compiler-generated constructor
```

Explicit Default Constructor

```
class Student
{
    public int Id;
    public string Name;

    // Explicitly defined default constructor
    public Student()
    {
        Id = 0;
        Name = "Unknown";
        Console.WriteLine("Student object created");
    }
}
```

When Compiler Doesn't Generate Default Constructor

> ⚠ **Important Rule**
>
> If you define **ANY** constructor (even with parameters), the compiler **WILL NOT** generate a default constructor!

```
class Student
{
    public int Id;
    public string Name;

    // We defined a parameterized constructor
    public Student(int id)
    {
        Id = id;
    }

    // Compiler does NOT generate default constructor
}

// Usage
var s1 = new Student(1);    // ✅ OK
var s2 = new Student();     // ❌ ERROR: No parameterless constructor
```

Solution: Explicitly add default constructor if needed:

```
class Student
{
    public int Id;
    public string Name;

    // Default constructor
    public Student()
    {
        Id = 0;
        Name = "Unknown";
    }

    // Parameterized constructor
    public Student(int id)
    {
        Id = id;
    }
}
```

```
// Now both work
var s1 = new Student();      // ✅ OK
var s2 = new Student(1);     // ✅ OK
```

Initialization Order

ⓘ What Happens During Default Constructor

```
class Example
{
    public int X = 10;           // 1. Field initializer runs first
    public int Y { get; set; } = 20;  // 2. Property initializer

    public Example()             // 3. Constructor body runs last
    {
        Console.WriteLine($"X={X}, Y={Y}"); // X=10, Y=20
    }
}
```

Example with Logic

```
class Logger
{
    private string logPath;
    private StreamWriter writer;

    // Default constructor with initialization logic
    public Logger()
    {
        logPath = $"log_{DateTime.Now:yyyyMMdd}.txt";
        writer = new StreamWriter(logPath, append: true);
        Console.WriteLine($"Logger initialized: {logPath}");
    }
}
```

Parameterized Constructor

Definition

A **parameterized constructor** accepts parameters to initialize object fields/properties with specific values during creation.

Basic Syntax

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

    // Parameterized constructor
    public Student(int id, string name, int age)
    {
        Id = id;
        Name = name;
        Age = age;
    }
}

// Usage
var student = new Student(1, "Ali", 20);
```

Multiple Parameters

```
class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
    public decimal Salary { get; set; }
    public DateTime HireDate { get; set; }

    public Employee(int id, string name, string dept, decimal salary, DateTime
hireDate)
    {
        Id = id;
        Name = name;
        Department = dept;
        Salary = salary;
        HireDate = hireDate;
    }
}
```

```
// Usage
var emp = new Employee(
    id: 101,
    name: "Sara",
    dept: "IT",
    salary: 75000m,
    hireDate: DateTime.Now
);
```

With Validation

✓ **Best Practice: Validate in Constructor**

```
class BankAccount
{
    public string AccountNumber { get; }
    public decimal Balance { get; private set; }

    public BankAccount(string accountNumber, decimal initialBalance)
    {
        // Validation
        if (string.IsNullOrWhiteSpace(accountNumber))
            throw new ArgumentException("Account number is required");

        if (initialBalance < 0)
            throw new ArgumentException("Initial balance cannot be
negative");

        // Initialization
        AccountNumber = accountNumber;
        Balance = initialBalance;
    }
}
```

Computed Values

```
class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
    public double Area { get; set; }
    public double Perimeter { get; set; }
```

```
    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;

        // Computed properties
        Area = width * height;
        Perimeter = 2 * (width + height);
    }
}
```

With Default Values in Constructor Body

```
class Configuration
{
    public string Host { get; set; }
    public int Port { get; set; }
    public bool UseSsl { get; set; }
    public int Timeout { get; set; }

    public Configuration(string host, int port)
    {
        Host = host;
        Port = port;

        // Set defaults for other properties
        UseSsl = true;
        Timeout = 30;
    }
}
```

Copy Constructor

> **Definition**
>
> A **copy constructor** creates a new object by copying the values from an existing object of the same type.

Basic Implementation

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

    // Regular constructor
    public Person(string name, int age, string city)
    {
        Name = name;
        Age = age;
        City = city;
    }

    // Copy constructor
    public Person(Person other)
    {
        Name = other.Name;
        Age = other.Age;
        City = other.City;
    }
}

// Usage
var person1 = new Person("Ali", 25, "Cairo");
var person2 = new Person(person1);  // Copy constructor

Console.WriteLine(person2.Name);  // Ali
```

Shallow Copy vs Deep Copy

Shallow Copy

> ⚠ **Shallow Copy Limitation**
>
> Copies reference types by reference, not by value!

```
class Address
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

```
class Person
{
    public string Name { get; set; }
    public Address Address { get; set; }

    // Shallow copy constructor
    public Person(Person other)
    {
        Name = other.Name;
        Address = other.Address;  // ⚠ Reference copied, not object
    }
}

// Problem
var person1 = new Person
{
    Name = "Ali",
    Address = new Address { Street = "Main St", City = "Cairo" }
};

var person2 = new Person(person1);  // Shallow copy
person2.Address.City = "Alexandria"; // Changes person1's address too!

Console.WriteLine(person1.Address.City);  // Alexandria (not Cairo!)
```

Deep Copy

✓ Deep Copy Solution

Create new instances of reference types

```
class Address
{
    public string Street { get; set; }
    public string City { get; set; }

    // Copy constructor for Address
    public Address(Address other)
    {
        Street = other.Street;
        City = other.City;
    }
}
```

```
class Person
{
    public string Name { get; set; }
    public Address Address { get; set; }

    // Deep copy constructor
    public Person(Person other)
    {
        Name = other.Name;

        // Create NEW Address object
        if (other.Address != null)
        {
            Address = new Address(other.Address);
        }
    }
}

// Now it works correctly
var person1 = new Person
{
    Name = "Ali",
    Address = new Address { Street = "Main St", City = "Cairo" }
};

var person2 = new Person(person1);  // Deep copy
person2.Address.City = "Alexandria"; // Only changes person2

Console.WriteLine(person1.Address.City);  // Cairo ✅
Console.WriteLine(person2.Address.City);  // Alexandria ✅
```

With Collections (Deep Copy)

```
class Student
{
    public string Name { get; set; }
    public List<int> Grades { get; set; }

    public Student(string name)
    {
        Name = name;
        Grades = new List<int>();
    }

    // Copy constructor with deep copy of list
```

```
    public Student(Student other)
    {
        Name = other.Name;

        // Create new list with copied values
        Grades = new List<int>(other.Grades);
    }
}

// Usage
var student1 = new Student("Ali");
student1.Grades.AddRange(new[] { 90, 85, 95 });

var student2 = new Student(student1);
student2.Grades.Add(100);  // Only affects student2

Console.WriteLine(student1.Grades.Count);  // 3
Console.WriteLine(student2.Grades.Count);  // 4
```

ICloneable Pattern

```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }

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

    // Copy constructor
    public Person(Person other)
    {
        Name = other.Name;
        Age = other.Age;
    }

    // ICloneable implementation
    public object Clone()
    {
        return new Person(this);  // Uses copy constructor
    }
}
```

```
// Usage
var person1 = new Person("Ali", 25);
var person2 = (Person)person1.Clone();
```

Static Constructor

Definition

A **static constructor** initializes static members of a class. It runs **once** per application domain, before any instance is created or static members are accessed.

Basic Syntax

```
class Configuration
{
    public static string AppName;
    public static int MaxConnections;

    // Static constructor
    static Configuration()
    {
        AppName = "MyApplication";
        MaxConnections = 100;
        Console.WriteLine("Static constructor called");
    }
}

// First access triggers static constructor
Console.WriteLine(Configuration.AppName);  // Static constructor called
                                           // MyApplication
```

Key Characteristics

Static Constructor Rules

1. **No access modifier** (not public, private, etc.)
2. **No parameters** (cannot be overloaded)
3. **Called automatically** by CLR
4. **Called once** per application domain

5. **Before first instance creation** or static member access
6. **Cannot call directly** from code
7. **No `this` keyword** (no instance context)

When Static Constructor Runs

```
class Logger
{
    public static string LogPath;

    static Logger()
    {
        LogPath = $"logs_{DateTime.Now:yyyyMMdd}.txt";
        Console.WriteLine("Static constructor executed");
    }

    public Logger()
    {
        Console.WriteLine("Instance constructor executed");
    }
}

// Execution order
var log1 = new Logger();
// Output:
// Static constructor executed    (only once)
// Instance constructor executed

var log2 = new Logger();
// Output:
// Instance constructor executed  (static NOT called again)
```

Visual Timeline

```
Application Starts
       ↓
First Access to Logger (new Logger() or Logger.LogPath)
       ↓
Static Constructor Runs (ONCE)
       ↓
Static Members Initialized
       ↓
Instance Constructor Runs
```

```
        ↓
Object Created
        ↓
Subsequent new Logger() calls
        ↓
Only Instance Constructor Runs (Static already done)
```

Practical Examples

Example 1: Configuration Loading

```
class AppSettings
{
    public static string ConnectionString;
    public static int CacheTimeout;
    public static bool EnableLogging;

    static AppSettings()
    {
        // Load from configuration file
        ConnectionString = LoadFromConfig("ConnectionString");
        CacheTimeout = int.Parse(LoadFromConfig("CacheTimeout"));
        EnableLogging = bool.Parse(LoadFromConfig("EnableLogging"));

        Console.WriteLine("Application settings loaded");
    }

    private static string LoadFromConfig(string key)
    {
        // Simulate loading from config file
        return ConfigurationManager.AppSettings[key];
    }
}
```

Example 2: Singleton Pattern

```
class Singleton
{
    private static readonly Singleton _instance;

    // Static constructor
    static Singleton()
    {
        _instance = new Singleton();
```

```
        Console.WriteLine("Singleton instance created");
    }

    // Private instance constructor
    private Singleton()
    {
        // Initialization
    }

    public static Singleton Instance => _instance;
}

// Usage
var instance1 = Singleton.Instance;  // Static constructor runs
var instance2 = Singleton.Instance;  // Same instance, no constructor
```

Example 3: Static Data Initialization

```
class MathConstants
{
    public static double Pi;
    public static double E;
    public static double GoldenRatio;

    static MathConstants()
    {
        Pi = 3.14159265359;
        E = 2.71828182846;
        GoldenRatio = 1.61803398875;

        Console.WriteLine("Math constants initialized");
    }
}
```

Static vs Instance Constructor

```
class Example
{
    public static int StaticField;
    public int InstanceField;

    // Static constructor
    static Example()
    {
```

```
        StaticField = 100;
        Console.WriteLine("Static constructor");
    }

    // Instance constructor
    public Example()
    {
        InstanceField = 200;
        Console.WriteLine("Instance constructor");
    }
}

// First object creation
var obj1 = new Example();
// Output:
// Static constructor
// Instance constructor

// Second object creation
var obj2 = new Example();
// Output:
// Instance constructor (static already ran)
```

Exception Handling

> **⚡ Static Constructor Exceptions**
>
> If a static constructor throws an exception, the type becomes unusable!

```
class ProblematicClass
{
    static ProblematicClass()
    {
        throw new Exception("Static constructor failed!");
    }

    public ProblematicClass() { }
}

// First attempt
try
{
    var obj1 = new ProblematicClass();
}
```

```
catch (TypeInitializationException ex)
{
    Console.WriteLine("First attempt failed");
}

// Second attempt
try
{
    var obj2 = new ProblematicClass();  // Also fails!
}
catch (TypeInitializationException ex)
{
    Console.WriteLine("Second attempt failed too!");
}
```

Private Constructor

> **Definition**
>
> A **private constructor** prevents object creation from outside the class. Used for utility classes, singletons, or controlling object instantiation.

Basic Usage

```
class Utility
{
    // Private constructor
    private Utility()
    {
        // Cannot be called from outside
    }

    public static void DoSomething()
    {
        Console.WriteLine("Static method");
    }
}

// Usage
// var util = new Utility();  // ❌ ERROR: Constructor is private
Utility.DoSomething();        // ✅ OK: Static method
```

Use Case 1: Utility/Helper Classes

✓ **Static-Only Classes**

Classes with only static members should have private constructors

```
class MathHelper
{
    // Private constructor prevents instantiation
    private MathHelper() { }

    public static int Add(int a, int b) => a + b;
    public static int Multiply(int a, int b) => a * b;
    public static double Average(params int[] numbers)
    {
        return numbers.Average();
    }
}

// Usage - only static methods
int result = MathHelper.Add(5, 10);
double avg = MathHelper.Average(1, 2, 3, 4, 5);
```

Use Case 2: Singleton Pattern

ⓘ **Thread-Safe Singleton**

Private constructor ensures only one instance exists

```
class DatabaseConnection
{
    private static DatabaseConnection _instance;
    private static readonly object _lock = new object();

    private string connectionString;

    // Private constructor
    private DatabaseConnection()
    {
        connectionString = "Server=localhost;Database=MyDB";
        Console.WriteLine("Database connection initialized");
    }
```

```
    public static DatabaseConnection Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new DatabaseConnection();
                    }
                }
            }
            return _instance;
        }
    }

    public void ExecuteQuery(string query)
    {
        Console.WriteLine($"Executing: {query}");
    }
}

// Usage
var db1 = DatabaseConnection.Instance;
var db2 = DatabaseConnection.Instance;  // Same instance
Console.WriteLine(db1 == db2);  // True
```

Use Case 3: Factory Pattern

Controlled Object Creation

Use private constructor with factory methods

```
class Employee
{
    public int Id { get; private set; }
    public string Name { get; private set; }
    public string Type { get; private set; }
    public decimal Salary { get; private set; }

    // Private constructor
```

```
    private Employee(int id, string name, string type, decimal salary)
    {
        Id = id;
        Name = name;
        Type = type;
        Salary = salary;
    }

    // Factory methods
    public static Employee CreateManager(int id, string name)
    {
        return new Employee(id, name, "Manager", 100000m);
    }

    public static Employee CreateDeveloper(int id, string name)
    {
        return new Employee(id, name, "Developer", 80000m);
    }

    public static Employee CreateIntern(int id, string name)
    {
        return new Employee(id, name, "Intern", 30000m);
    }
}

// Usage
var manager = Employee.CreateManager(1, "Ali");
var developer = Employee.CreateDeveloper(2, "Sara");
var intern = Employee.CreateIntern(3, "Omar");

// Cannot create directly
// var emp = new Employee(4, "Ahmed", "Designer", 50000);  // ❌ ERROR
```

Use Case 4: Builder Pattern

```
class Pizza
{
    public string Size { get; private set; }
    public List<string> Toppings { get; private set; }
    public bool ExtraCheese { get; private set; }

    // Private constructor
    private Pizza()
    {
        Toppings = new List<string>();
```

```
    }

    // Nested Builder class
    public class Builder
    {
        private Pizza pizza = new Pizza();

        public Builder SetSize(string size)
        {
            pizza.Size = size;
            return this;
        }

        public Builder AddTopping(string topping)
        {
            pizza.Toppings.Add(topping);
            return this;
        }

        public Builder WithExtraCheese()
        {
            pizza.ExtraCheese = true;
            return this;
        }

        public Pizza Build()
        {
            return pizza;
        }
    }
}

// Usage
var pizza = new Pizza.Builder()
    .SetSize("Large")
    .AddTopping("Pepperoni")
    .AddTopping("Mushrooms")
    .WithExtraCheese()
    .Build();
```

Use Case 5: Constants Class

```
class Constants
{
    // Private constructor prevents instantiation
```

```
    private Constants() { }

    public const string AppName = "MyApp";
    public const int MaxRetries = 3;
    public const double TaxRate = 0.15;

    public static readonly DateTime AppStartTime = DateTime.Now;
}

// Usage
Console.WriteLine(Constants.AppName);
Console.WriteLine(Constants.MaxRetries);
```

Combining Private and Public Constructors

```
class Counter
{
    public int Value { get; private set; }

    // Private constructor for internal use
    private Counter(int initialValue)
    {
        Value = initialValue;
    }

    // Public factory methods
    public static Counter StartFromZero()
    {
        return new Counter(0);
    }

    public static Counter StartFrom(int value)
    {
        if (value < 0)
            throw new ArgumentException("Value must be non-negative");

        return new Counter(value);
    }

    public void Increment() => Value++;
}

// Usage
var counter1 = Counter.StartFromZero();
var counter2 = Counter.StartFrom(100);
```

Constructor Chaining

Definition

Constructor chaining is calling one constructor from another constructor using `this()` or `base()` keyword. It reduces code duplication.

Chaining Within Same Class (this)

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

    // Constructor 1: Full parameters
    public Student(int id, string name, int age, string department)
    {
        Id = id;
        Name = name;
        Age = age;
        Department = department;
        Console.WriteLine("Full constructor called");
    }

    // Constructor 2: Chains to Constructor 1
    public Student(int id, string name, int age)
        : this(id, name, age, "General")  // Calls full constructor
    {
        Console.WriteLine("3-parameter constructor called");
    }

    // Constructor 3: Chains to Constructor 2
    public Student(int id, string name)
        : this(id, name, 18)  // Calls 3-parameter constructor
    {
        Console.WriteLine("2-parameter constructor called");
    }

    // Constructor 4: Chains to Constructor 3
    public Student(string name)
```

```
        : this(0, name)  // Calls 2-parameter constructor
    {
        Console.WriteLine("1-parameter constructor called");
    }
}

// Usage
var student = new Student("Ali");
// Output:
// Full constructor called
// 3-parameter constructor called
// 2-parameter constructor called
// 1-parameter constructor called
```

Execution Order

> **Constructor Chain Execution**
>
> The **chained** constructor executes **BEFORE** the calling constructor's body
>
> ```
> Constructor Chain:
> this(params) → Chained constructor body → Current constructor body
> ```

```
class Example
{
    public int X { get; set; }

    public Example(int x)
    {
        X = x;
        Console.WriteLine($"1. Constructor with param: X = {X}");
    }

    public Example() : this(100)
    {
        Console.WriteLine($"2. Default constructor: X = {X}");
    }
}

var obj = new Example();
// Output:
```

```
// 1. Constructor with param: X = 100  (runs first)
// 2. Default constructor: X = 100     (runs second)
```

Chaining to Base Class (base)

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

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
        Console.WriteLine("Person constructor");
    }
}

class Employee : Person
{
    public int EmployeeId { get; set; }
    public string Department { get; set; }

    // Chain to base class constructor
    public Employee(int id, string name, int age, string dept)
        : base(name, age)  // Calls Person constructor
    {
        EmployeeId = id;
        Department = dept;
        Console.WriteLine("Employee constructor");
    }
}

// Usage
var emp = new Employee(1, "Ali", 25, "IT");
// Output:
// Person constructor
// Employee constructor
```

Complex Chaining Example

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

```
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
    public bool IsActive { get; set; }

    // Primary constructor
    public Product(int id, string name, decimal price, string category, bool
isActive)
    {
        Id = id;
        Name = name;
        Price = price;
        Category = category;
        IsActive = isActive;
    }

    // Chain 1: Set default IsActive
    public Product(int id, string name, decimal price, string category)
        : this(id, name, price, category, isActive: true)
    {
    }

    // Chain 2: Set default Category
    public Product(int id, string name, decimal price)
        : this(id, name, price, category: "General")
    {
    }

    // Chain 3: Set default Price
    public Product(int id, string name)
        : this(id, name, price: 0m)
    {
    }

    // Chain 4: Set default Name
    public Product(int id)
        : this(id, name: "Unnamed Product")
    {
    }
}

// Usage - all valid
var p1 = new Product(1, "Laptop", 999.99m, "Electronics", true);
var p2 = new Product(2, "Mouse", 29.99m, "Accessories");
var p3 = new Product(3, "Keyboard", 59.99m);
```

```
var p4 = new Product(4, "Monitor");
var p5 = new Product(5);
```

With Default Parameters

```
class Configuration
{
    public string Host { get; set; }
    public int Port { get; set; }
    public bool UseSsl { get; set; }

    // Constructor with default parameters
    public Configuration(string host, int port = 80, bool useSsl = false)
    {
        Host = host;
        Port = port;
        UseSsl = useSsl;
    }

    // Chain with different defaults
    public Configuration(string host, bool useSsl)
        : this(host, port: useSsl ? 443 : 80, useSsl: useSsl)
    {
    }
}

// Usage
var config1 = new Configuration("localhost");
var config2 = new Configuration("localhost", 8080);
var config3 = new Configuration("localhost", true);  // Uses SSL, port 443
```

Benefits of Constructor Chaining

✓ **Advantages**

- ✅ **DRY Principle**: Don't Repeat Yourself
- ✅ **Single Initialization Point**: One place for main logic
- ✅ **Easier Maintenance**: Change once, affects all
- ✅ **Cleaner Code**: Less duplication
- ✅ **Consistent Initialization**: Same validation/logic

Constructor Overloading

Definition

Constructor overloading means having multiple constructors in the same class with different parameter lists. Each provides a different way to create objects.

Basic Example

```
class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }

    // Constructor 1: No parameters (unit square)
    public Rectangle()
    {
        Width = 1;
        Height = 1;
    }

    // Constructor 2: One parameter (square)
    public Rectangle(double size)
    {
        Width = size;
        Height = size;
    }

    // Constructor 3: Two parameters (rectangle)
    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }
}

// Usage - different ways to create
var r1 = new Rectangle();           // 1x1 square
var r2 = new Rectangle(5);          // 5x5 square
var r3 = new Rectangle(4, 6);       // 4x6 rectangle
```

Rules for Overloading

ⓘ Overloading Requirements

1. **Different parameter count**, OR
2. **Different parameter types**, OR
3. **Different parameter order**

Return type and parameter names don't matter!

```
class Example
{
    // ✅ Valid: Different parameter count
    public Example() { }
    public Example(int x) { }
    public Example(int x, int y) { }

    // ✅ Valid: Different parameter types
    public Example(string s) { }
    public Example(double d) { }

    // ✅ Valid: Different parameter order
    public Example(int x, string s) { }
    public Example(string s, int x) { }

    // ❌ Invalid: Same signature
    // public Example(int a) { }  // ERROR: Already have Example(int)

    // ❌ Invalid: Parameter names don't matter
    // public Example(int number) { }  // ERROR: Same as Example(int x)
}
```

Practical Example: Date Class

```
class Date
{
    public int Year { get; set; }
    public int Month { get; set; }
    public int Day { get; set; }

    // Constructor 1: Full date
    public Date(int year, int month, int day)
    {
        Year = year;
        Month = month;
```

```
        Day = day;
    }

    // Constructor 2: From DateTime
    public Date(DateTime dateTime)
    {
        Year = dateTime.Year;
        Month = dateTime.Month;
        Day = dateTime.Day;
    }

    // Constructor 3: From string (parse)
    public Date(string dateString)
    {
        var dt = DateTime.Parse(dateString);
        Year = dt.Year;
        Month = dt.Month;
        Day = dt.Day;
    }

    // Constructor 4: Today
    public Date() : this(DateTime.Now)
    {
    }
}

// Usage - multiple ways to create dates
var date1 = new Date(2024, 1, 15);
var date2 = new Date(DateTime.Now);
var date3 = new Date("2024-01-15");
var date4 = new Date();  // Today
```

With Different Types

```
class Temperature
{
    public double Value { get; set; }
    public string Unit { get; set; }

    // Constructor 1: Celsius
    public Temperature(double celsius)
    {
        Value = celsius;
        Unit = "Celsius";
    }
```

```
    // Constructor 2: Fahrenheit (bool flag to differentiate)
    public Temperature(double fahrenheit, bool isFahrenheit)
    {
        if (isFahrenheit)
        {
            Value = (fahrenheit - 32) * 5 / 9;  // Convert to Celsius
            Unit = "Celsius";
        }
    }

    // Constructor 3: Kelvin (string to differentiate)
    public Temperature(double kelvin, string unit)
    {
        if (unit == "Kelvin")
        {
            Value = kelvin - 273.15;  // Convert to Celsius
            Unit = "Celsius";
        }
    }
}

// Usage
var temp1 = new Temperature(25);                    // Celsius
var temp2 = new Temperature(77, isFahrenheit: true); // Fahrenheit
var temp3 = new Temperature(298.15, "Kelvin");      // Kelvin
```

Combining Overloading with Chaining

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

    // Main constructor
    public Person(string firstName, string lastName, int age, string email)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
        Email = email;
    }
```

```
    // Overload 1: No email
    public Person(string firstName, string lastName, int age)
        : this(firstName, lastName, age, string.Empty)
    {
    }

    // Overload 2: No age or email
    public Person(string firstName, string lastName)
        : this(firstName, lastName, 18, string.Empty)
    {
    }

    // Overload 3: Full name as one string
    public Person(string fullName)
    {
        var parts = fullName.Split(' ');
        FirstName = parts[0];
        LastName = parts.Length > 1 ? parts[1] : string.Empty;
        Age = 18;
        Email = string.Empty;
    }
}
```

Object Initializer Alternative

Modern Alternative

With properties, object initializers can reduce need for many constructors:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; } = 18;
    public string Email { get; set; }

    // Just one or two constructors needed
    public Person() { }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

```
// Usage with object initializer
var person = new Person
{
    FirstName = "Ali",
    LastName = "Ahmed",
    Email = "ali@example.com"
    // Age uses default value 18
};
```

Primary Constructor (C# 12)

Definition

Primary constructors (C# 12) allow you to declare constructor parameters directly in the class declaration, making them available throughout the class.

Basic Syntax

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

    public Student(int id, string name)
    {
        Id = id;
        Name = name;
    }
}

// Primary constructor way (C# 12)
class Student(int id, string name)
{
    public int Id { get; set; } = id;
    public string Name { get; set; } = name;
}
```

Detailed Explanation

I already covered this extensively in the previous file! Check the **Primary Constructors (C# 12)** section in the "Advanced Features" document for:

- Full syntax
- Parameter scope
- Usage patterns
- Combining with other features
- Best practices

Expression-Bodied Constructors

> **Definition**
>
> **Expression-bodied constructors** use lambda syntax `=>` for simple single-expression constructor bodies.

Basic Syntax

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

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

// Expression-bodied constructor (C# 7.0+)
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
```

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

With Single Statement

```
class Logger
{
    private string filePath;

    // Expression-bodied constructor
    public Logger(string path) => filePath = path ?? "default.log";
}
```

With Tuple Deconstruction

```
class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double width, double height) => (Width, Height) = (width,
height);
}
```

Multiple Assignments

```
class Config
{
    public string Host { get; set; }
    public int Port { get; set; }
    public bool Secure { get; set; }

    public Config(string host, int port) =>
        (Host, Port, Secure) = (host, port, port == 443);
}
```

Limitations

> ⚠ **When NOT to Use**
>
> Expression-bodied constructors are for **simple** cases only:
>
> ❌ Cannot use for:

- Multiple statements
- Complex validation
- Exception throwing
- Console output
- Method calls (usually)

✅ Best for:

- Simple property assignments
- Single field initialization

```
// ❌ Don't do this - too complex
public Person(string name, int age) =>
    (Name, Age) = ValidateAndAssign(name, age);  // Too complex

// ✅ Do this instead - traditional
public Person(string name, int age)
{
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException("Name required");

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

    Name = name;
    Age = age;
}
```

Constructor with Optional Parameters

Definition

Optional parameters allow constructors to have default values, reducing the need for multiple overloaded constructors.

Basic Syntax

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

    // Constructor with optional parameters
    public Student(int id = 0, string name = "Unknown", int age = 18)
    {
        Id = id;
        Name = name;
        Age = age;
    }
}

// Usage - all valid
var s1 = new Student();                    // All defaults
var s2 = new Student(1);                   // id=1, rest default
var s3 = new Student(1, "Ali");            // id=1, name="Ali", age=18
var s4 = new Student(1, "Ali", 20);        // All specified
var s5 = new Student(id: 2, age: 25);      // Skip name parameter
```

Rules for Optional Parameters

> **Important Rules**
>
> 1. **Optional parameters must come last** (after required parameters)
> 2. **Default value must be compile-time constant**
> 3. **Can skip parameters using named arguments**

```
// ✅ Correct
public Person(string name, int age = 18, string city = "Cairo") { }

// ❌ Wrong - optional before required
public Person(int age = 18, string name) { }

// ❌ Wrong - non-constant default
public Person(string name, DateTime date = DateTime.Now) { }
// Use this instead:
public Person(string name, DateTime date = default)
{
```

```
    date = date == default ? DateTime.Now : date;
}
```

Complex Example

```
class HttpRequest
{
    public string Url { get; set; }
    public string Method { get; set; }
    public int Timeout { get; set; }
    public Dictionary<string, string> Headers { get; set; }
    public string Body { get; set; }

    public HttpRequest(
        string url,
        string method = "GET",
        int timeout = 30,
        Dictionary<string, string> headers = null,
        string body = null)
    {
        Url = url;
        Method = method;
        Timeout = timeout;
        Headers = headers ?? new Dictionary<string, string>();
        Body = body;
    }
}

// Usage
var req1 = new HttpRequest("https://api.example.com");
var req2 = new HttpRequest("https://api.example.com", "POST");
var req3 = new HttpRequest(
    url: "https://api.example.com",
    method: "PUT",
    timeout: 60,
    body: "{\"data\": \"value\"}"
);
```

With Validation

```
class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
```

```
    public int Stock { get; set; }

    public Product(
        string name,
        decimal price = 0,
        int stock = 0)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new ArgumentException("Name is required");

        if (price < 0)
            throw new ArgumentException("Price cannot be negative");

        if (stock < 0)
            throw new ArgumentException("Stock cannot be negative");

        Name = name;
        Price = price;
        Stock = stock;
    }
}
```

Benefits vs Overloading

✓ Optional Parameters vs Constructor Overloading

Optional Parameters:

```
// One constructor
public Person(string name, int age = 18, string city = "Cairo") { }
```

Constructor Overloading:

```
// Multiple constructors
public Person(string name)
    : this(name, 18, "Cairo") { }

public Person(string name, int age)
    : this(name, age, "Cairo") { }

public Person(string name, int age, string city)
{
    // Implementation
}
```

Use Optional Parameters when:

- ✅ Simple default values
- ✅ Most parameters are optional
- ✅ Want named argument flexibility

Use Overloading when:

- ✅ Different initialization logic needed
- ✅ Complex parameter relationships
- ✅ Better IntelliSense experience

Constructor with Named Parameters

Definition

Named parameters allow you to specify arguments by parameter name rather than position, improving readability and flexibility.

Basic Usage

```
class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Department { get; set; }
    public decimal Salary { get; set; }

    public Employee(string name, int age, string department, decimal salary)
    {
        Name = name;
        Age = age;
        Department = department;
        Salary = salary;
    }
}

// Without named parameters (positional)
var emp1 = new Employee("Ali", 25, "IT", 75000);
```

```
// With named parameters (any order!)
var emp2 = new Employee(
    name: "Sara",
    age: 30,
    salary: 85000,
    department: "HR"
);

// Mix positional and named
var emp3 = new Employee("Omar", 28, department: "Sales", salary: 70000);
```

Benefits

✓ Advantages of Named Parameters

1. **Clarity**: Obvious what each argument represents
2. **Flexibility**: Can specify in any order
3. **Skip optionals**: Easy to skip optional parameters
4. **Maintainability**: Less error-prone
5. **Self-documenting**: Code reads like documentation

With Optional Parameters

```
class Configuration
{
    public string Host { get; set; }
    public int Port { get; set; }
    public bool UseSsl { get; set; }
    public int Timeout { get; set; }

    public Configuration(
        string host,
        int port = 80,
        bool useSsl = false,
        int timeout = 30)
    {
        Host = host;
        Port = port;
        UseSsl = useSsl;
        Timeout = timeout;
    }
}
```

```
// Named parameters make it clear
var config1 = new Configuration(
    host: "localhost",
    useSsl: true,  // Skip port, use default 80
    timeout: 60
);

// Without named parameters - confusing!
var config2 = new Configuration("localhost", 80, true, 60);
```

Complex Real-World Example

```
class EmailMessage
{
    public string From { get; set; }
    public string To { get; set; }
    public string Subject { get; set; }
    public string Body { get; set; }
    public bool IsHtml { get; set; }
    public int Priority { get; set; }
    public List<string> Attachments { get; set; }

    public EmailMessage(
        string from,
        string to,
        string subject,
        string body,
        bool isHtml = false,
        int priority = 3,
        List<string> attachments = null)
    {
        From = from;
        To = to;
        Subject = subject;
        Body = body;
        IsHtml = isHtml;
        Priority = priority;
        Attachments = attachments ?? new List<string>();
    }
}

// Very readable with named parameters
var email = new EmailMessage(
    from: "sender@example.com",
    to: "recipient@example.com",
```

```
    subject: "Important Update",
    body: "<h1>Hello!</h1>",
    isHtml: true,
    priority: 1
);

// Compare to positional - hard to read
var email2 = new EmailMessage(
    "sender@example.com",
    "recipient@example.com",
    "Important Update",
    "<h1>Hello!</h1>",
    true,
    1,
    null
);
```

Struct Constructors

Definition

Structs are value types that can have constructors, but with different rules than classes.

Key Differences from Classes

Struct Constructor Rules

1. **Cannot have parameterless constructor** (before C# 10)
2. **Must initialize ALL fields** in constructor
3. **Cannot call `this()` before initializing fields**
4. **Cannot have field initializers** (before C# 10)
5. **Default constructor always exists** (sets all to default)

Basic Struct Constructor

```
struct Point
{
    public int X;
    public int Y;
```

```
    // Must initialize ALL fields
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}

// Usage
var p1 = new Point(10, 20);
var p2 = default(Point);  // X=0, Y=0 (default constructor)
```

Struct with Properties

```
struct Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public double Area => Width * Height;
}
```

C# 10+ Features

> **ⓘ C# 10 Improvements**
>
> C# 10 relaxed some struct restrictions:

```
struct Point
{
    public int X { get; set; } = 0;     // ✅ Field initializers allowed
    public int Y { get; set; } = 0;

    // ✅ Parameterless constructor allowed
    public Point()
    {
```

```
        X = 1;
        Y = 1;
    }

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

Readonly Struct

```
readonly struct ImmutablePoint
{
    public int X { get; }
    public int Y { get; }

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

Record Struct (C# 10)

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

// Usage
var p1 = new Point(10, 20);
var p2 = p1 with { X = 30 };  // Creates copy with modification
```

Record Constructors

Definition

Records have special constructor behavior with automatic property initialization and deconstruction.

Positional Record

```
// Automatic constructor created
record Person(string Name, int Age);

// Equivalent to:
record Person
{
    public string Name { get; init; }
    public int Age { get; init; }

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

    public void Deconstruct(out string name, out int age)
    {
        name = Name;
        age = Age;
    }
}
```

Custom Constructor in Record

```
record Person(string Name, int Age)
{
    // Additional constructor
    public Person(string name) : this(name, 18)
    {
    }

    // Validation in primary constructor
    public string Name { get; init; } =
        !string.IsNullOrWhiteSpace(Name) ? Name : throw new
ArgumentException();
}
```

See the previous "Advanced Features" document for complete record details!

Constructor Best Practices

✓ **Best Practices Summary**

1. Keep Constructors Simple

```
// ✅ Good - Simple initialization
public Person(string name, int age)
{
    Name = name;
    Age = age;
}

// ❌ Bad - Too much logic
public Person(string name, int age)
{
    Name = name;
    Age = age;
    LoadFromDatabase();
    SendWelcomeEmail();
    UpdateStatistics();
    // Too much work!
}
```

2. Validate Parameters

```
public BankAccount(string accountNumber, decimal balance)
{
    if (string.IsNullOrWhiteSpace(accountNumber))
        throw new ArgumentException("Account number required");

    if (balance < 0)
        throw new ArgumentException("Balance cannot be negative");

    AccountNumber = accountNumber;
    Balance = balance;
}
```

3. Use Constructor Chaining

```
// ✅ Good - DRY principle
public Person(string name, int age, string city)
{
    Name = name;
    Age = age;
```

```
    City = city;
}

public Person(string name, int age) : this(name, age, "Unknown") { }
public Person(string name) : this(name, 18) { }
```

4. Consider Factory Methods

```
// For complex creation logic
public class User
{
    private User(string username, string hashedPassword)
    {
        Username = username;
        HashedPassword = hashedPassword;
    }

    public static User Create(string username, string plainPassword)
    {
        var hashed = HashPassword(plainPassword);
        return new User(username, hashed);
    }
}
```

5. Prefer Immutability

```
// ✅ Good - Immutable
public class Point
{
    public int X { get; }
    public int Y { get; }

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

6. Use Optional Parameters Wisely

```
// ✅ Good - Simple defaults
public Logger(string path = "app.log", bool append = true) { }
```

```
// ❌ Bad - Too many optional parameters
public DataProcessor(
    string input = null,
    string output = null,
    bool compress = false,
    bool encrypt = false,
    int threads = 4,
    long maxSize = 1000000,
    string format = "json"
) { }  // Consider builder pattern instead
```

7. Document Complex Constructors

```
/// <summary>
/// Creates a new database connection
/// </summary>
/// <param name="connectionString">Database connection string</param>
/// <param name="timeout">Command timeout in seconds (default: 30)</param>
/// <param name="pooling">Enable connection pooling (default: true)</param>
public DatabaseConnection(
    string connectionString,
    int timeout = 30,
    bool pooling = true)
{
    // Implementation
}
```

Summary Table

Constructor Type	Use Case	Key Feature
Default	Simple initialization	No parameters
Parameterized	Custom initialization	Accepts parameters
Copy	Clone objects	Takes same type
Static	Class-level init	Runs once per type
Private	Singleton, factory	Restrict instantiation
Chaining (this)	Reuse logic	Call another constructor
Chaining (base)	Inheritance	Call parent constructor

Constructor Type	Use Case	Key Feature
Overloaded	Multiple creation ways	Different signatures
Primary (C# 12)	Concise syntax	Parameters in declaration
Expression-bodied	Simple one-liner	Lambda syntax
Optional params	Flexible creation	Default values
Named params	Clear intent	Specify by name

Additional Resources

Learning Resources

Official Documentation

- Constructors (C# Programming Guide)
- Instance Constructors
- Static Constructors
- Primary Constructors

End of Documentation

Document Info

Created: January 2026
Topic: Complete Guide to C# Constructors
Tags: `#csharp` `#constructors` `#oop` `#initialization` `#best-practices`

By Abdullah Ali

Contact : +201012613453