Polymorphism, Object Class & Collections in C# (.NET 8)

Advanced Reference & Training Guide 🔧 📚

Table of Contents

POLYMORPHISM

Definition & Theory

Polymorphism (Greek: "many forms") is the ability of objects to take multiple forms. In C#, polymorphism allows a single interface to represent different underlying data types. This is one of the four pillars of OOP and enables flexible, extensible designs.

Two Flavors of Polymorphism

Type	When	Mechanism	Performance
Compile-time	Known at compile time	Method overloading, operator overloading	Zero runtime cost
Runtime	Known only at runtime	Virtual methods, abstract classes, interfaces	Virtual dispatch overhead

---

Compile-time Polymorphism

1) Method Overloading

Definition: Multiple methods with the *same name* but *different parameters* in the same class.

Syntax & Rules:

- Methods must differ by: number of parameters, type of parameters, or order of parameters
- Return type alone does NOT distinguish methods
- Helps API clarity—callers use one method name for related operations

Example: Simple

```
public class Calculator
{
    public int Add(int a, int b) => a + b;

    public double Add(double a, double b) => a + b;

    public int Add(int a, int b, int c) => a + b + c;

    public string Add(string a, string b) => a + b;
}

// Usage
var calc = new Calculator();
Console.WriteLine(calc.Add(5, 10));                 // 15 (int overload)
Console.WriteLine(calc.Add(5.5, 10.5));             // 16.0 (double overload)
Console.WriteLine(calc.Add(5, 10, 15));             // 30 (three-param overload)
Console.WriteLine(calc.Add("Hello", " World"));     // "Hello World" (string overload)
```

Example: Complex—Practical Banking

```csharp
public class BankAccount
{
    private decimal _balance;

    // Withdraw with reason and logging
    public void Withdraw(decimal amount)
    {
        if (amount <= 0) throw new ArgumentException("Amount must be positive");
        _balance -= amount;
        Console.WriteLine($"Withdrew {amount}");
    }

    public void Withdraw(decimal amount, string reason)
    {
        Withdraw(amount);  // reuse
        Console.WriteLine($"Reason: {reason}");
    }

    public void Withdraw(decimal amount, string reason, DateTime date)
    {
        Withdraw(amount, reason);  // reuse
        Console.WriteLine($"Date: {date:yyyy-MM-dd}");
    }

    // Overload for withdrawal by check number
    public void Withdraw(decimal amount, int checkNumber)
    {
        Withdraw(amount);
        Console.WriteLine($"Check #{checkNumber}");
    }
}

// Usage
var account = new BankAccount();
account.Withdraw(100);                                    // simple
account.Withdraw(100, "ATM withdrawal");                  // with reason
account.Withdraw(100, "Check payment", DateTime.Now);    // full details
account.Withdraw(100, 1001);                              // by check
```

Pro Tips:

- Use overloading to simplify API—customers don't memorize many method names
- Order parameters from most common to least common usage
- Consider using default parameters or builder patterns for many variants

Pitfalls:

- Too many overloads obscure intent; limit to 3–5 per method family
- Implicit type conversions can cause unexpected overload selection
- Overloading with similar numeric types (int vs long) can be confusing

---

2) Operator Overloading

Definition: Redefine how built-in operators (+, -, *, ==, [], etc.) work with custom types.

Syntax:

```
public static ReturnType operator OP (Type left [, Type right])
{
    // implementation
}
```

Supported Operators:

- Unary: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`
- Binary: `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `&`, `|`, `^`, `<<`, `>>`
- Indexer: `[]`
- Conversion: implicit/explicit casts

Example: Money/Currency Class

```
public class Money : IComparable<Money>
{
    public decimal Amount { get; }
    public string Currency { get; }

    public Money(decimal amount, string currency = "USD")
    {
        Amount = amount;
        Currency = currency;
    }

    // Addition
    public static Money operator +(Money left, Money right)
    {
        if (left.Currency != right.Currency)
            throw new InvalidOperationException("Cannot add different currencies");
        return new Money(left.Amount + right.Amount, left.Currency);
    }

    // Subtraction
    public static Money operator -(Money left, Money right)
    {
        if (left.Currency != right.Currency)
            throw new InvalidOperationException("Cannot subtract different currencies");
        return new Money(left.Amount - right.Amount, left.Currency);
    }

    // Scalar multiplication
    public static Money operator *(Money left, decimal multiplier)
    {
        return new Money(left.Amount * multiplier, left.Currency);
    }

    public static Money operator *(decimal multiplier, Money right) => right * multiplier;

    // Division by scalar
    public static Money operator /(Money left, decimal divisor)
    {
        return new Money(left.Amount / divisor, left.Currency);
    }

    // Equality
    public static bool operator ==(Money left, Money right)
    {
        return left.Amount == right.Amount && left.Currency == right.Currency;
    }

    public static bool operator !=(Money left, Money right) => !(left == right);

    // Comparison
    public static bool operator <(Money left, Money right)
    {
        if (left.Currency != right.Currency)
            throw new InvalidOperationException("Cannot compare different currencies");
```

```
        return left.Amount < right.Amount;
    }

    public static bool operator >(Money left, Money right)
    {
        if (left.Currency != right.Currency)
            throw new InvalidOperationException("Cannot compare different currencies");
        return left.Amount > right.Amount;
    }

    public static bool operator <=(Money left, Money right) => left < right || left ==
right;
    public static bool operator >=(Money left, Money right) => left > right || left ==
right;

    // Explicit conversion to decimal
    public static explicit operator decimal(Money money) => money.Amount;

    // Implicit conversion from decimal
    public static implicit operator Money(decimal amount) => new Money(amount);

    public override string ToString() => $"{Amount:C} {Currency}";

    public override bool Equals(object? obj) => obj is Money m && this == m;
    public override int GetHashCode() => Amount.GetHashCode() ^ Currency.GetHashCode();

    public int CompareTo(Money? other) => other == null ? 1 :
Amount.CompareTo(other.Amount);
}

// Usage
var price1 = new Money(100, "USD");
var price2 = new Money(50, "USD");

var total = price1 + price2;          // 150 USD
var discount = price1 - price2;       // 50 USD
var doubled = price1 * 2;             // 200 USD
var quarter = price1 / 4;             // 25 USD

Console.WriteLine(price1 > price2);   // true
Console.WriteLine(price1 == price1);  // true

Money implicitMoney = 75.50m;         // implicit conversion
decimal amount = (decimal)price1;     // explicit conversion
```

Pro Tips:

- Always override `Equals()` and `GetHashCode()` when overloading `==`
- Keep operator implementations simple and intuitive
- Document non-obvious behavior

Pitfalls:

- Violating semantic meaning (e.g., `+` should combine, not subtract)
- Operator chaining with side effects is confusing

- Overloading bitwise operators on non-flag types is rarely useful

---

Runtime Polymorphism

1) Method Overriding (Virtual/Override)

Definition: A derived class replaces the implementation of a base class method marked `virtual`.

Keywords:

- `virtual` in base class: marks method as overrideable
- `override` in derived class: provides new implementation
- `new` in derived class: hides base method (breaks polymorphism—avoid)

Syntax:

```
public class Base
{
    public virtual void DoSomething() { /* base impl */ }
}

public class Derived : Base
{
    public override void DoSomething() { /* new impl */ }
}
```

Example: Payment Processing

```csharp
public abstract class PaymentProcessor
{
    protected decimal Amount { get; }

    protected PaymentProcessor(decimal amount) => Amount = amount;

    public virtual bool Validate()
    {
        return Amount > 0;
    }

    public virtual void Process()
    {
        if (!Validate())
            throw new InvalidOperationException("Invalid payment");
        Console.WriteLine($"Processing payment of {Amount}");
    }

    public virtual void PrintReceipt()
    {
        Console.WriteLine("=== Receipt ===");
        Console.WriteLine($"Amount: {Amount}");
    }
}

public class CreditCardProcessor : PaymentProcessor
{
    public string CardNumber { get; }

    public CreditCardProcessor(decimal amount, string cardNumber) : base(amount)
    {
        CardNumber = cardNumber;
    }

    public override bool Validate()
    {
        // Call base validation first
        if (!base.Validate()) return false;

        // Credit card specific validation
        return CardNumber.Length == 16 && CardNumber.All(char.IsDigit);
    }

    public override void Process()
    {
        Console.WriteLine("Processing via Credit Card...");
        base.Process();  // call base
        Console.WriteLine($"Card: {CardNumber[^4..].PadLeft(16, '*')}");
    }

    public override void PrintReceipt()
    {
        base.PrintReceipt();
        Console.WriteLine($"Method: Credit Card (*{CardNumber[^4..]})");
```

```csharp
    }
}

public class PayPalProcessor : PaymentProcessor
{
    public string Email { get; }

    public PayPalProcessor(decimal amount, string email) : base(amount)
    {
        Email = email;
    }

    public override bool Validate()
    {
        return base.Validate() && Email.Contains("@");
    }

    public override void Process()
    {
        Console.WriteLine("Processing via PayPal...");
        base.Process();
        Console.WriteLine($"Email: {Email}");
    }

    public override void PrintReceipt()
    {
        base.PrintReceipt();
        Console.WriteLine($"Method: PayPal ({Email})");
    }
}

public class CheckProcessor : PaymentProcessor
{
    public string CheckNumber { get; }

    public CheckProcessor(decimal amount, string checkNumber) : base(amount)
    {
        CheckNumber = checkNumber;
    }

    public override void Process()
    {
        Console.WriteLine("Processing check...");
        base.Process();
        Console.WriteLine($"Check #: {CheckNumber}");
    }

    public override void PrintReceipt()
    {
        base.PrintReceipt();
        Console.WriteLine($"Method: Check #{CheckNumber}");
    }
}

// Polymorphic usage
```

```csharp
public class PaymentService
{
    public void ProcessPayments(List<PaymentProcessor> processors)
    {
        foreach (var processor in processors)
        {
            if (processor.Validate())
            {
                processor.Process();
                processor.PrintReceipt();
            }
            else
            {
                Console.WriteLine("Validation failed for processor");
            }
            Console.WriteLine();
        }
    }
}

// Usage
var service = new PaymentService();
var payments = new List<PaymentProcessor>
{
    new CreditCardProcessor(100, "1234567890123456"),
    new PayPalProcessor(50, "user@example.com"),
    new CheckProcessor(75, "12345")
};

service.ProcessPayments(payments);
```

Output:

```
Processing via Credit Card...
Processing payment of 100
Card: ****3456
=== Receipt ===
Amount: 100
Method: Credit Card (*3456)

Processing via PayPal...
Processing payment of 50
Email: user@example.com
=== Receipt ===
Amount: 50
Method: PayPal (user@example.com)

Processing check...
Processing payment of 75
Check #: 12345
=== Receipt ===
Amount: 75
Method: Check #12345
```

Pro Tips:

- Always call `base.Method()` when you want to extend (not replace) behavior

- Use `virtual` sparingly—overrides can be surprising to maintainers
- Sealed classes prevent further overriding (use when design is final)

Pitfalls:

- Calling virtual methods in constructors can invoke derived implementations before derived fields are initialized
- Using `new` instead of `override` hides the base method, breaking polymorphism
- Deep inheritance hierarchies become hard to reason about

---

2) Abstract Classes and Abstract Methods

Definition: A class that *cannot be instantiated* and *requires derived classes* to implement abstract members.

When to Use: Shared code + enforced contract (partial vs. full implementation).

Syntax:

```
public abstract class Base
{
    public abstract void RequiredMethod();  // no body

    public virtual void Optional() { }      // default body

    public void Concrete() { }              // normal method
}

public class Derived : Base
{
    public override void RequiredMethod() { /* must implement */ }
}
```

Example: E-commerce Order System

```
public abstract class OrderProcessor
{
    protected List<string> Items { get; } = new();

    // Abstract method—subclass MUST implement
    public abstract void ValidateOrder();

    // Abstract method for order-specific processing
    public abstract void ApplyDiscount();

    // Concrete method—shared by all subclasses
    public void AddItem(string item)
    {
        Items.Add(item);
    }

    // Template method pattern—defines structure
    public void Process()
    {
        Console.WriteLine("Order Processing Started...");
        ValidateOrder();
        ApplyDiscount();
        Ship();
        Console.WriteLine("Order Processed.");
    }

    protected virtual void Ship()
    {
        Console.WriteLine("Shipping order with standard method.");
    }
}

public class RetailOrder : OrderProcessor
{
    public decimal Total { get; private set; }

    public override void ValidateOrder()
    {
        if (Items.Count == 0)
            throw new InvalidOperationException("Order has no items");
        Console.WriteLine("Validating retail order...");
    }

    public override void ApplyDiscount()
    {
        // Retail: 10% discount if 3+ items
        if (Items.Count >= 3)
        {
            Console.WriteLine("Applied 10% bulk discount.");
            Total *= 0.9m;
        }
    }
}
```

```csharp
public class WholesaleOrder : OrderProcessor
{
    public decimal Total { get; private set; }

    public override void ValidateOrder()
    {
        if (Items.Count < 10)
            throw new InvalidOperationException("Wholesale requires minimum 10 items");
        Console.WriteLine("Validating wholesale order...");
    }

    public override void ApplyDiscount()
    {
        // Wholesale: 25% discount
        Console.WriteLine("Applied 25% wholesale discount.");
        Total *= 0.75m;
    }

    protected override void Ship()
    {
        Console.WriteLine("Shipping via bulk freight.");
    }
}

public class SubscriptionOrder : OrderProcessor
{
    public override void ValidateOrder()
    {
        Console.WriteLine("Validating subscription order...");
    }

    public override void ApplyDiscount()
    {
        Console.WriteLine("Applied subscription discount (5%).");
    }

    protected override void Ship()
    {
        Console.WriteLine("Subscription auto-ships monthly.");
    }
}

// Usage
OrderProcessor retail = new RetailOrder();
retail.AddItem("Book");
retail.AddItem("Pen");
retail.AddItem("Notebook");
retail.Process();

Console.WriteLine("\n---\n");

OrderProcessor wholesale = new WholesaleOrder();
for (int i = 0; i < 15; i++)
    wholesale.AddItem($"Item{i}");
wholesale.Process();
```

```
// Cannot do: OrderProcessor proc = new OrderProcessor(); // ❌ Cannot instantiate abstract class
```

Pro Tips:

- Use abstract classes when subclasses *share code* and require enforced contracts
- Combine abstract methods with concrete helper methods for *template method pattern*
- Abstract classes can have fields, properties, private methods—not just contracts

Pitfalls:

- Mixing abstract and concrete methods confuses the design intent
- Too many abstract methods bloats derived classes

Interfaces

Definition: Pure contract—specifies *what* a type must do, not *how*.

Characteristics:

- No fields (until C# 8.0, can have default implementations)
- All members are public by default (no private/protected)
- A class can implement multiple interfaces
- Since C# 8.0, interfaces can have default implementations (but keep minimal)

Syntax:

```
public interface IInterface
{
    void RequiredMethod();
    string Property { get; set; }

    // C# 8+: default implementation (use sparingly)
    void DefaultMethod() { Console.WriteLine("Default"); }
}
```

Example: Multi-interface Implementation

```csharp
// Role-based interfaces
public interface IEmployee
{
    string Name { get; }
    void Work();
}

public interface IManager
{
    void Approve(string request);
    List<IEmployee> GetTeam();
}

public interface ITrainable
{
    void AttendTraining(string course);
}

public class Developer : IEmployee, IManager, ITrainable
{
    public string Name { get; }
    private List<IEmployee> _team = new();
    private List<string> _trainings = new();

    public Developer(string name)
    {
        Name = name;
    }

    // IEmployee implementation
    public void Work()
    {
        Console.WriteLine($"{Name} is writing code...");
    }

    // IManager implementation
    public void Approve(string request)
    {
        Console.WriteLine($"{Name} approved: {request}");
    }

    public List<IEmployee> GetTeam() => _team;

    // ITrainable implementation
    public void AttendTraining(string course)
    {
        _trainings.Add(course);
        Console.WriteLine($"{Name} attended {course}");
    }
}

// Polymorphic usage with interfaces
var dev = new Developer("Alice");
dev.Work();                          // as IEmployee
```

```
dev.Approve("New Feature");          // as IManager
dev.AttendTraining("C# Async");      // as ITrainable

// Work with as specific interface
IEmployee emp = dev;
emp.Work();

IManager mgr = dev;
mgr.Approve("Bug Fix");

// Multiple dispatch
List<IEmployee> employees = new() { dev };
foreach (var e in employees) e.Work();
```

Pro Tips:

- Interfaces = contracts; use for *behavior abstraction* and *dependency injection*
- Follow Interface Segregation Principle (ISP): small, focused interfaces
- Avoid default implementations—interfaces are for contracts, not behavior

Pitfalls:

- Interfaces with many methods are hard to implement (violates ISP)
- Default implementations in interfaces break contract semantics

---

Abstract Classes vs Interfaces: When to Use Each

Aspect	Abstract Class	Interface
Purpose	Partial implementation + contract	Pure contract
State	Can have fields	No fields (C# 8+ readonly; not recommended)
Constructors	Can have custom logic	No constructors
Access Modifiers	Can be private, protected	Always public (conceptually)
Inheritance vs Implementation	One base class inheritance	Multiple interfaces Implementation
Implementation	Subclass inherits code	Class provides all impl
Version Stability	Hard to add new members	Easier (default impl since C# 8)
Example	`Shape`, `Animal`	`IComparable`, `IDisposable`

Decision Tree

```
Does the type have shared implementation code?
├─ YES → Abstract Class (share code, enforce contract)
└─ NO  → Interface (pure contract)

Is it describing behavior multiple unrelated types should have?
└─ YES → Interface (e.g., IComparable, IDisposable)

Is it "IS-A" relationship?
└─ YES → Abstract Class (Dog IS-A Animal)

Is it "CAN-DO" behavior?
└─ YES → Interface (Person CAN-DO ITrainable)
```

Example: Deciding Between Abstract and Interface

```csharp
// Bad: Interface with too much shared behavior
public interface IShape
{
    double GetArea() { return 0; }  // default impl hides intent
    double GetPerimeter() { return 0; }
}

// Good: Abstract class for shared geometric behavior
public abstract class Shape
{
    protected double Side { get; set; }

    public abstract double GetArea();
    public abstract double GetPerimeter();
}

// Good: Interface for independent behavior
public interface IDrawable
{
    void Draw();
}

public class Circle : Shape, IDrawable
{
    private double Radius { get; set; }

    public override double GetArea() => Math.PI * Radius * Radius;
    public override double GetPerimeter() => 2 * Math.PI * Radius;

    public void Draw()
    {
        Console.WriteLine("Drawing circle...");
    }
}
```

OBJECT CLASS AND ADVANCED CONCEPTS

System.Object Class

Definition: The base class of *all* types in C#. Every class implicitly inherits from `object`.

Key Point: Even value types (struct, int, bool) inherit from `object` via boxing.

Core Members:

Member	Purpose
`ToString()`	String representation
`Equals(object)`	Value equality comparison
`GetHashCode()`	Hash code for collections
`GetType()`	Runtime type information
`ReferenceEquals(obj1, obj2)`	Reference equality

Diagram: Object Inheritance Hierarchy

```
              object
                ↑
                |
        ┌───────┼───────┐
        |       |       |
    ValueType  Type1       Type2
        |
        ├── int, double, bool, etc.
        └── struct, enum
```

ToString()

Definition: Returns a string representation of an object.

Default Behavior: Returns the fully qualified type name (e.g., `"MyNamespace.MyClass"`).

Override for Clarity: Custom implementations make debugging and logging easier.

Syntax:

```
public override string ToString() => /* return string representation */
```

Example: Comprehensive

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

    // Default implementation (not overridden)
    public string DefaultToString => base.ToString();  // "ConsoleApp.Person"

    // Simple override
    public override string ToString() => $"{FirstName} {LastName}";
}

public class Employee : Person
{
    public string EmployeeId { get; set; }
    public decimal Salary { get; set; }

    // Detailed override
    public override string ToString()
    {
        return $"[EMP-{EmployeeId}] {FirstName} {LastName}, Age: {Age}, Email: {Email}, 
Salary: ${Salary:N2}";
    }
}

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

    public override string ToString()
    {
        return $"{Street}, {City}, {ZipCode}";
    }
}

// Usage
var person = new Person { FirstName = "John", LastName = "Doe", Age = 30 };
Console.WriteLine(person);  // "John Doe"

var emp = new Employee
{
    FirstName = "Jane",
    LastName = "Smith",
    Age = 28,
    Email = "jane@example.com",
    EmployeeId = "E001",
    Salary = 75000
};
Console.WriteLine(emp);  // "[EMP-E001] Jane Smith, Age: 28, Email: jane@example.com, 
Salary: $75,000.00"
```

```
var addr = new Address { Street = "123 Main St", City = "Springfield", ZipCode = "12345"
};
Console.WriteLine(addr);  // "123 Main St, Springfield, 12345"
```

Pro Tips:

- Use for logging, debugging, and user-friendly output
- Format for readability (e.g., include labels like "Age: 30")
- Keep ToString() lightweight—avoid expensive operations

Pitfalls:

- Very long ToString() with concatenation is slow
- Exposing internal structure via ToString() can be a security risk

---

Equals() and GetHashCode()

Definition:

- `Equals()`: Determines if two objects are equal in value
- `GetHashCode()`: Generates a hash code for use in hash-based collections

Critical Rule: If you override `Equals()`, you *must* override `GetHashCode()` to maintain the contract: *if two objects are equal, they must have the same hash code*.

Equals() Example

Default Behavior: Reference equality (object.ReferenceEquals).

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

    // Value-based equality (two Products with same ID are equal)
    public override bool Equals(object? obj)
    {
        // Check if null or different type
        if (obj == null || obj.GetType() != GetType())
            return false;

        // Cast and compare
        var other = (Product)obj;
        return ProductId == other.ProductId;  // equality based on ID
    }

    // MUST override GetHashCode() when overriding Equals()
    public override int GetHashCode()
    {
        return ProductId.GetHashCode();
    }

    public override string ToString() => $"{Name} ({ProductId}): ${Price}";
}

// Usage
var p1 = new Product { ProductId = "P001", Name = "Laptop", Price = 999 };
var p2 = new Product { ProductId = "P001", Name = "Laptop", Price = 999 };
var p3 = new Product { ProductId = "P002", Name = "Mouse", Price = 25 };

Console.WriteLine(p1.Equals(p2));  // true (same ProductId)
Console.WriteLine(p1 == p2);       // false (different references)
Console.WriteLine(p1.Equals(p3));  // false (different ProductId)

// In hash-based collection
var set = new HashSet<Product> { p1, p2, p3 };
Console.WriteLine(set.Count);  // 2 (p1 and p2 are considered equal)
```

GetHashCode() Best Practices

```csharp
public class Order
{
    public string OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public List<string> Items { get; set; } = new();

    // Simple hash code (immutable key)
    public override int GetHashCode()
    {
        return OrderId.GetHashCode();
    }

    public override bool Equals(object? obj)
    {
        if (obj is not Order other) return false;
        return OrderId == other.OrderId;
    }
}

public class Invoice
{
    public string InvoiceId { get; set; }
    public decimal Amount { get; set; }

    // Composite hash code (multiple properties)
    public override int GetHashCode()
    {
        unchecked  // prevent overflow exceptions
        {
            int hash = 17;
            hash = hash * 31 + InvoiceId.GetHashCode();
            hash = hash * 31 + Amount.GetHashCode();
            return hash;
        }
    }

    // Or use C# 7.0+ tuple hash
    public override int GetHashCode() => HashCode.Combine(InvoiceId, Amount);

    public override bool Equals(object? obj)
    {
        if (obj is not Invoice other) return false;
        return InvoiceId == other.InvoiceId && Amount == other.Amount;
    }
}
```

Pro Tips:

- Use `HashCode.Combine()` (C# 7.0+) for simple, correct composite hashes
- Hash codes do NOT need to be unique (collisions are okay, just avoid excessive ones)
- Only hash on *immutable* properties; mutable properties cause bugs in collections

Pitfalls:

- Overriding `Equals()` without `GetHashCode()` breaks HashSet, Dictionary
- Changing hash code after adding to a collection loses the object
- Using mutable fields for equality is a common bug

Equality: == vs Equals()

== Operator:

- Calls `operator==` if defined, else uses reference equality
- For reference types, default is reference equality
- For value types, uses value equality

Equals() Method:

- Virtual method; can be overridden
- For reference types, default is reference equality
- Can define custom logic

Best Practice: For custom classes, override both `==` and `Equals()` consistently.

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

    // Override Equals()
    public override bool Equals(object? obj)
    {
        if (obj is not Account other) return false;
        return AccountNumber == other.AccountNumber;
    }

    public override int GetHashCode() => AccountNumber.GetHashCode();

    // Override == operator
    public static bool operator ==(Account? left, Account? right)
    {
        if (ReferenceEquals(left, right)) return true;
        if (left is null || right is null) return false;
        return left.Equals(right);
    }

    public static bool operator !=(Account? left, Account? right) => !(left == right);
}

// Usage
var acc1 = new Account { AccountNumber = "ACC001", Balance = 1000 };
var acc2 = new Account { AccountNumber = "ACC001", Balance = 1000 };

Console.WriteLine(acc1 == acc2);        // true (now consistent)
Console.WriteLine(acc1.Equals(acc2));  // true
```

GetType() and typeof

GetType():

- Runtime method; returns Type of instance
- Used for reflection and runtime type checks

typeof:

- Compile-time operator; returns Type of a type
- Cannot be used on instances directly

```
public class Animal { }
public class Dog : Animal { }

// Usage
var dog = new Dog();

// GetType() - instance method
Type t1 = dog.GetType();  // typeof(Dog)

// typeof - compile-time operator
Type t2 = typeof(Dog);

Console.WriteLine(t1 == t2);  // true

// Practical example: dynamic behavior
if (dog.GetType() == typeof(Dog))
{
    Console.WriteLine("It's a dog!");
}

// Polymorphic check
Animal animal = dog;
if (animal.GetType() == typeof(Dog))  // checks actual type, not declared
{
    Console.WriteLine("This variable holds a Dog instance");
}

// Reflection
var methods = typeof(Dog).GetMethods();
var properties = dog.GetType().GetProperties();
```

Garbage Collection in .NET

Definition: Automatic memory management that reclaims objects no longer referenced.

How It Works:

1. Mark phase: Mark reachable objects (follow references from roots)
2. Sweep phase: Free unmarked objects
3. Compact phase: Reorganize heap (reduces fragmentation)

Generations: Generational GC optimizes collection based on object age

Generation	Age	Collection Frequency
Gen 0	Newly created	Frequent (fast)
Gen 1	Survived 1 collection	Moderate
Gen 2	Survived 2+ collections	Infrequent (full)

Example: GC in Action

```
public class Resource
{
    public string Name { get; set; }

    public Resource(string name)
    {
        Name = name;
        Console.WriteLine($"{Name} created");
    }

    ~Resource()  // Destructor (finalizer)
    {
        Console.WriteLine($"{Name} destroyed by GC");
    }
}

// GC demonstration
var res1 = new Resource("Res1");  // allocated on heap
var res2 = new Resource("Res2");

res1 = null;  // res1 now eligible for collection
res2 = res2;  // res2 still referenced

GC.Collect();  // Force collection (normally automatic)
GC.WaitForPendingFinalizers();

Console.WriteLine("GC completed");
// Output:
// Res1 created
// Res2 created
// Res1 destroyed by GC
// GC completed
```

Pro Tips:

- Let GC do its job; manually forcing collection (GC.Collect()) is rarely needed
- Use `using` statements for deterministic cleanup of resources

Pitfalls:

- Finalizers (destructors) slow down GC; prefer `IDisposable`
- Event subscriptions without unsubscribe prevent GC (memory leaks)

IDisposable and using Statement

Definition: Pattern for deterministic cleanup of unmanaged resources (file handles, database connections, etc.).

Interface:

```csharp
public interface IDisposable
{
    void Dispose();
}
```

Typical Pattern (Pre-C# 8):

```csharp
public class FileManager : IDisposable
{
    private FileStream? _fileStream;
    private bool _disposed = false;

    public void OpenFile(string path)
    {
        _fileStream = new FileStream(path, FileMode.Open);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (_disposed) return;

        if (disposing)
        {
            // Dispose managed resources
            _fileStream?.Dispose();
        }

        // Dispose unmanaged resources here if any

        _disposed = true;
    }

    ~FileManager()
    {
        Dispose(false);
    }
}
```

Modern Pattern (C# 8+):

```csharp
public class DatabaseConnection : IDisposable
{
    private SqlConnection? _connection;

    public void Connect(string connectionString)
    {
        _connection = new SqlConnection(connectionString);
        _connection.Open();
    }

    public void Dispose()
    {
        _connection?.Dispose();
        GC.SuppressFinalize(this);
    }
}
```

Using Statement:

```csharp
// Old: using block
using (var manager = new FileManager())
{
    manager.OpenFile("data.txt");
    // ...
}  // Dispose() called automatically

// Modern: using declaration (C# 8+)
using var connection = new DatabaseConnection();
connection.Connect("Server=localhost");
// ...
// Dispose() called automatically at end of scope
```

Example: Real-world Resource Management

```csharp
public class TempFileManager : IDisposable
{
    private string _tempFilePath;

    public TempFileManager()
    {
        _tempFilePath = Path.GetTempFileName();
        Console.WriteLine($"Created temp file: {_tempFilePath}");
    }

    public void WriteData(string data)
    {
        File.WriteAllText(_tempFilePath, data);
    }

    public string ReadData() => File.ReadAllText(_tempFilePath);

    public void Dispose()
    {
        if (File.Exists(_tempFilePath))
        {
            File.Delete(_tempFilePath);
            Console.WriteLine($"Deleted temp file: {_tempFilePath}");
        }
    }
}

// Usage
using var manager = new TempFileManager();
manager.WriteData("Hello, World!");
Console.WriteLine(manager.ReadData());
// Dispose() is called automatically when exiting the using block
```

Pro Tips:

- Use `using` statements to ensure `Dispose()` is always called, even if exceptions occur
- Implement both `Dispose()` and a finalizer only if you have unmanaged resources
- C# 8+ `using` declaration is cleaner than `using` blocks

Pitfalls:

- Forgetting to implement `IDisposable` for types that hold unmanaged resources
- Not calling `Dispose()` on objects that need cleanup
- Disposing objects you don't own

Finalizers and Destructors

Definition: Special methods invoked by the GC to clean up before object is destroyed.

Syntax:

```csharp
public class MyClass
{
    ~MyClass()  // Finalizer / Destructor
    {
        // cleanup code
    }
}
```

Key Points:

- Finalizers are *not* deterministic—called by GC at unknown time
- Only use if managing unmanaged resources directly
- Finalizers add GC overhead—prefer `IDisposable` instead

Example: Avoiding Finalizers

```csharp
// ❌ Bad: Using finalizer
public class BadResourceManager
{
    private IntPtr _unManagedHandle;

    public BadResourceManager()
    {
        _unManagedHandle = AllocateUnmanagedResource();
    }

    ~BadResourceManager()  // Slow, unpredictable
    {
        FreeUnmanagedResource(_unManagedHandle);
    }

    private IntPtr AllocateUnmanagedResource() => /* P/Invoke call */;
    private void FreeUnmanagedResource(IntPtr handle) => /* P/Invoke call */;
}

// ✅ Good: Using IDisposable + SafeHandle
public class GoodResourceManager : IDisposable
{
    private SafeFileHandle? _handle;
    private bool _disposed = false;

    public GoodResourceManager()
    {
        _handle = new SafeFileHandle(IntPtr.Zero, true);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (_disposed) return;

        if (disposing)
        {
            _handle?.Dispose();
        }

        _disposed = true;
    }
}
```

COLLECTIONS FRAMEWORK

Overview of Collections

Definition: Specialized classes for storing and managing groups of objects.

Hierarchy:

```
                  IEnumerable
                      ↑
              ┌────────┴────────┐
              │                 │
        ICollection          IEnumerable<T>
              ↑                 ↑
        ┌─────┴─────┐          │
        │           │          │
      List<T>   Dictionary  ICollection<T>
        │           ↑          │
        │           │      ┌───┴───┐
        │       Hashtable  │       │
        │                Queue<T> Stack<T>
    ArrayList
```

Generic Collections

Why Generics?

- Type-safe: catch errors at compile time
- No boxing/unboxing overhead
- Better performance
- Intellisense support

1) List

Definition: Ordered, mutable collection of type-safe items (resizable array).

Common Methods:

```
var list = new List<int> { 1, 2, 3 };
list.Add(4);                 // Add single item
list.AddRange(new[] {5, 6});  // Add multiple
list.Insert(0, 0);           // Insert at index
list.Remove(2);              // Remove by value
list.RemoveAt(0);            // Remove by index
list.Contains(1);            // Check existence
list.IndexOf(2);             // Get index of item
list.Count;                  // Number of items
list.Clear();                // Remove all items
```

Example: Product Inventory

```csharp
public class Product
{
    public string Id { get; set; }
    public string Name { get; set; }
    public int Quantity { get; set; }
    public decimal Price { get; set; }

    public override string ToString() => $"{Name} (Q: {Quantity}, Price: ${Price})";
}

public class Inventory
{
    private List<Product> _products = new();

    public void AddProduct(Product product)
    {
        if (_products.Any(p => p.Id == product.Id))
            throw new InvalidOperationException("Product already exists");
        _products.Add(product);
    }

    public void RemoveProduct(string productId)
    {
        _products.RemoveAll(p => p.Id == productId);
    }

    public Product? FindProduct(string productId)
    {
        return _products.Find(p => p.Id == productId);
    }

    public IEnumerable<Product> GetLowStockProducts(int threshold)
    {
        return _products.Where(p => p.Quantity < threshold);
    }

    public void DisplayAll()
    {
        foreach (var product in _products)
            Console.WriteLine(product);
    }
}

// Usage
var inventory = new Inventory();
inventory.AddProduct(new Product { Id = "P001", Name = "Laptop", Quantity = 5, Price = 999
});
inventory.AddProduct(new Product { Id = "P002", Name = "Mouse", Quantity = 50, Price = 25
});
inventory.AddProduct(new Product { Id = "P003", Name = "Keyboard", Quantity = 2, Price =
75 });

inventory.DisplayAll();
```

```
Console.WriteLine("\nLow stock:");
foreach (var p in inventory.GetLowStockProducts(10))
    Console.WriteLine(p);
```

---

2) Dictionary<TKey, TValue>

Definition: Unordered key-value pairs (hash table). Key must be unique.

Common Methods:

```
var dict = new Dictionary<string, int> { { "key1", 1 }, { "key2", 2 } };
dict["key3"] = 3;                       // Add/update
dict.Add("key4", 4);                    // Add (throws if exists)
dict.Remove("key1");                    // Remove by key
dict.ContainsKey("key2");               // Check key exists
dict.ContainsValue(2);                  // Check value exists
dict["key2"];                           // Access by key
dict.TryGetValue("key2", out var val); // Safe access
dict.Keys;                              // Get all keys
dict.Values;                            // Get all values
```

Example: Student Grades

```
public class GradeBook
{
    private Dictionary<string, List<int>> _studentGrades = new();

    public void AddGrade(string studentName, int grade)
    {
        if (!_studentGrades.ContainsKey(studentName))
            _studentGrades[studentName] = new List<int>();

        _studentGrades[studentName].Add(grade);
    }

    public double GetAverageGrade(string studentName)
    {
        if (!_studentGrades.ContainsKey(studentName))
            return 0;

        return _studentGrades[studentName].Average();
    }

    public void DisplayReport()
    {
        foreach (var kvp in _studentGrades)
        {
            var avgGrade = kvp.Value.Average();
            Console.WriteLine($"{kvp.Key}: {avgGrade:F2} ({string.Join(", ",
kvp.Value)})");
        }
    }
}

// Usage
var gradeBook = new GradeBook();
gradeBook.AddGrade("Alice", 95);
gradeBook.AddGrade("Alice", 87);
gradeBook.AddGrade("Bob", 78);
gradeBook.AddGrade("Bob", 85);

gradeBook.DisplayReport();
// Output:
// Alice: 91.00 (95, 87)
// Bob: 81.50 (78, 85)
```

---

3) Queue

Definition: FIFO (First-In-First-Out) collection.

Common Methods:

```csharp
var queue = new Queue<string>();
queue.Enqueue("first");      // Add to back
queue.Dequeue();             // Remove from front
queue.Peek();                // View front without removing
queue.Count;                 // Number of items
```

Example: Print Job Queue

```csharp
public class PrintQueue
{
    private Queue<string> _jobs = new();

    public void SubmitJob(string jobName)
    {
        _jobs.Enqueue(jobName);
        Console.WriteLine($"Job submitted: {jobName}");
    }

    public void ProcessNextJob()
    {
        if (_jobs.Count == 0)
        {
            Console.WriteLine("No jobs to process");
            return;
        }

        var job = _jobs.Dequeue();
        Console.WriteLine($"Processing: {job}");
    }

    public void ShowQueue()
    {
        if (_jobs.Count == 0)
        {
            Console.WriteLine("Queue is empty");
            return;
        }

        Console.WriteLine($"Queue ({_jobs.Count} jobs):");
        foreach (var job in _jobs)
            Console.WriteLine($"  - {job}");
    }
}

// Usage
var printQueue = new PrintQueue();
printQueue.SubmitJob("Document1.pdf");
printQueue.SubmitJob("Document2.pdf");
printQueue.SubmitJob("Document3.pdf");
printQueue.ShowQueue();

printQueue.ProcessNextJob();
printQueue.ProcessNextJob();
printQueue.ShowQueue();
```

---

4) Stack

Definition: LIFO (Last-In-First-Out) collection.

Common Methods:

```
var stack = new Stack<string>();
stack.Push("item1");          // Add to top
stack.Pop();                  // Remove from top
stack.Peek();                 // View top without removing
stack.Count;                  // Number of items
```

Example: Undo/Redo Functionality

```csharp
public class TextEditor
{
    private string _content = "";
    private Stack<string> _undoStack = new();
    private Stack<string> _redoStack = new();

    public void Type(string text)
    {
        _undoStack.Push(_content);
        _redoStack.Clear();  // Clear redo on new action
        _content += text;
        Console.WriteLine($"Content: '{_content}'");
    }

    public void Undo()
    {
        if (_undoStack.Count == 0)
        {
            Console.WriteLine("Nothing to undo");
            return;
        }

        _redoStack.Push(_content);
        _content = _undoStack.Pop();
        Console.WriteLine($"Undo: '{_content}'");
    }

    public void Redo()
    {
        if (_redoStack.Count == 0)
        {
            Console.WriteLine("Nothing to redo");
            return;
        }

        _undoStack.Push(_content);
        _content = _redoStack.Pop();
        Console.WriteLine($"Redo: '{_content}'");
    }
}

// Usage
var editor = new TextEditor();
editor.Type("Hello");       // Hello
editor.Type(" World");      // Hello World
editor.Undo();              // Hello
editor.Undo();              // (empty)
editor.Redo();              // Hello
```

Non-Generic Collections

⚠️ **Legacy:** Pre-.NET 2.0, no type safety. Avoid in new code.

ArrayList

Definition: Untyped list (like List but without type safety).

```
var list = new ArrayList { 1, "two", 3.0, true };  // Mixed types!

// ❌ Requires casting
int num = (int)list[0];  // Risky: can throw InvalidCastException

// Better: Use List<object> or List<T>
var typedList = new List<object> { 1, "two", 3.0, true };
```

Hashtable

Definition: Untyped dictionary (like Dictionary<TKey, TValue> but without type safety).

```
var table = new Hashtable { { "key1", 1 }, { "key2", "two" } };  // Mixed types!

// ❌ Requires casting
int val = (int)table["key1"];  // Risky

// Better: Use Dictionary<TKey, TValue>
var dict = new Dictionary<string, object> { { "key1", 1 }, { "key2", "two" } };
```

Collection Patterns & Best Practices

LINQ Integration

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Filtering
var evens = numbers.Where(n => n % 2 == 0);  // 2, 4, 6, 8, 10

// Transforming
var squared = numbers.Select(n => n * n);    // 1, 4, 9, 16, 25, ...

// Ordering
var descending = numbers.OrderByDescending(n => n);

// Aggregation
var sum = numbers.Sum();     // 55
var avg = numbers.Average(); // 5.5
var max = numbers.Max();     // 10
var count = numbers.Count(n => n > 5);  // 5
```

Collection Initializers

```csharp
// Old way
var list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

// Modern: Collection initializer
var list = new List<int> { 1, 2, 3 };

// With objects
var products = new List<Product>
{
    new Product { Id = "P1", Name = "Laptop" },
    new Product { Id = "P2", Name = "Mouse" }
};

// Dictionary initializer
var cache = new Dictionary<string, int>
{
    { "key1", 100 },
    { "key2", 200 },
    ["key3"] = 300  // Modern syntax
};
```

Practical Application: Complete Shopping System

```
public class ShoppingCart
{
    private Dictionary<string, CartItem> _items = new();

    public class CartItem
    {
        public string ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public int Quantity { get; set; }

        public decimal Subtotal => UnitPrice * Quantity;
    }

    public void AddItem(string productId, string name, decimal price, int qty = 1)
    {
        if (_items.ContainsKey(productId))
        {
            _items[productId].Quantity += qty;
        }
        else
        {
            _items[productId] = new CartItem
            {
                ProductId = productId,
                ProductName = name,
                UnitPrice = price,
                Quantity = qty
            };
        }
    }

    public void RemoveItem(string productId)
    {
        _items.Remove(productId);
    }

    public decimal GetTotal() => _items.Values.Sum(item => item.Subtotal);

    public void DisplayCart()
    {
        if (_items.Count == 0)
        {
            Console.WriteLine("Cart is empty");
            return;
        }

        Console.WriteLine("=== Shopping Cart ===");
        foreach (var item in _items.Values)
        {
            Console.WriteLine($"{item.ProductName} (ID: {item.ProductId})");
            Console.WriteLine($"  Price: ${item.UnitPrice} x {item.Quantity} =
```

```
${item.Subtotal}");
        }
        Console.WriteLine($"Total: ${GetTotal():F2}");
    }
}

// Usage
var cart = new ShoppingCart();
cart.AddItem("P001", "Laptop", 999, 1);
cart.AddItem("P002", "Mouse", 25, 2);
cart.AddItem("P003", "USB Cable", 15, 3);

cart.DisplayCart();
// Output:
// === Shopping Cart ===
// Laptop (ID: P001)
//   Price: $999 x 1 = $999
// Mouse (ID: P002)
//   Price: $25 x 2 = $50
// USB Cable (ID: P003)
//   Price: $15 x 3 = $45
// Total: $1,094.00
```

Summary Table: When to Use Each Collection

Collection	Use Case	Performance
List	Ordered, indexed access	O(1) access, O(n) insert/remove
Dictionary<TK, TV>	Key-value lookups	O(1) average access
Queue	FIFO processing	O(1) enqueue/dequeue
Stack	LIFO processing, undo/redo	O(1) push/pop
HashSet	Unique items, fast membership	O(1) average add/remove/contains
LinkedList	Frequent insertions/removals	O(1) if position known, O(n) search
SortedList	Keep items sorted	O(log n) search, O(n) insert

Pro Tips & Best Practices 💡

1. **Always use generics:** List over ArrayList, Dictionary<K,V> over Hashtable
2. **Choose by access pattern:** Need indexed access? List. Key lookup? Dictionary. FIFO? Queue.
3. **LINQ is your friend:** Use Where, Select, GroupBy for filtering and transforming
4. **Initialize efficiently:** Use collection initializers for cleaner code
5. **Immutable where possible:** Use ImmutableList, ImmutableDictionary<K,V> for thread-safe scenarios

---

Common Pitfalls ⚠️

- Modifying collection while iterating (use ToList() to snapshot)

- Using ArrayList/Hashtable (non-generic) in new code
- Forgetting null checks when accessing Dictionary values
- Not accounting for O(n) operations in loops (use better data structures)
- Assuming Queue/Stack are sorted (they're not)

File completed with comprehensive coverage of Polymorphism, Object Class, and Collections Framework in C# .NET 8.