

# Ultimate cheat sheet

---

## C# Base Classes Guide

### 1. string

- **Purpose:** Represents a sequence of Unicode characters.
- **Initialization:**

```
csharp Copy Edit  
  
string name = "Alice";  
string greeting = string.Concat("Hello, ", name);  
string formatted = $"Hello, {name}";
```

- **Common Methods:**
  - Length, ToUpper(), ToLower(), Split(), Replace()

---

### 2. int (alias for System.Int32 )

- **Purpose:** Represents a 32-bit signed integer.
- **Initialization:**

```
csharp Copy Edit  
  
int age = 25;  
int sum = age + 5;
```

- **Notes:** Use TryParse or Parse to convert strings to int.

---

### 3. bool (alias for System.Boolean )

- **Purpose:** Represents a boolean value ( true or false ).

- **Initialization:**

```
csharp Copy Edit

bool isActive = true;
bool result = (5 > 3); // true
```

---

## 4. char (alias for System.Char )

- **Purpose:** Represents a single Unicode character.
- **Initialization:**

```
csharp Copy Edit

char letter = 'A';
char digit = '1';
```

---

## 5. double / float / decimal

- **Purpose:** Represent floating-point numbers with different precision.
- **Initialization:**

```
csharp Copy Edit

double d = 3.14159;
float f = 3.14f;
decimal money = 19.99m;
```

---

## 6. object

- **Purpose:** Base class of all types.

- **Initialization:**

csharp

Copy Edit

```
object obj = "Hello"; // Can store any type
```

---

## Collections

### 7. List<T>

- **Purpose:** Represents a dynamic array.
- **Namespace:** System.Collections.Generic
- **Initialization:**

csharp

Copy Edit

```
List<string> names = new List<string> { "Alice", "Bob" };  
names.Add("Charlie");
```

- **Common Methods:** Add(), Remove(), Contains(), Count, Sort()

---

### 8. Dictionary<TKey, TValue>

- **Purpose:** Represents a key-value pair collection.
- **Namespace:** System.Collections.Generic
- **Initialization:**

csharp

Copy Edit

```
Dictionary<string, int> ages = new Dictionary<string, int>  
{  
    { "Alice", 25 },  
    { "Bob", 30 }  
};  
int age = ages["Alice"];
```

---

### 9. HashSet<T>

- **Purpose:** Represents a collection of unique elements.
- **Namespace:** `System.Collections.Generic`
- **Initialization:**

csharp

Copy Edit

```
HashSet<int> uniqueNumbers = new HashSet<int> { 1, 2, 3 };  
uniqueNumbers.Add(4);
```

---

## 10. Queue<T>

- **Purpose:** FIFO (First-In, First-Out) collection.
- **Initialization:**

csharp

Copy Edit

```
Queue<string> queue = new Queue<string>();  
queue.Enqueue("First");  
string next = queue.Dequeue();
```

---

## 11. Stack<T>

- **Purpose:** LIFO (Last-In, First-Out) collection.
- **Initialization:**

csharp

Copy Edit

```
Stack<string> stack = new Stack<string>();  
stack.Push("Top");  
string last = stack.Pop();
```

---

## 12. Tuple<T1, T2, ...>

- **Purpose:** Groups multiple values into a single object.

- **Initialization:**

csharp

Copy Edit

```
var person = Tuple.Create("Alice", 25);  
string name = person.Item1;  
int age = person.Item2;
```

---

## 13. KeyValuePair<TKey, TValue>

- **Purpose:** Used with dictionaries or pair-like structures.
- **Initialization:**

csharp

Copy Edit

```
var pair = new KeyValuePair<string, int>("Alice", 25);
```

---

## 14. DateTime

- **Purpose:** Represents date and time.
- **Initialization:**

csharp

Copy Edit

```
DateTime now = DateTime.Now;  
DateTime birthday = new DateTime(2000, 5, 15);
```

---

## 15. TimeSpan

- **Purpose:** Represents a time interval.
- **Initialization:**

csharp

Copy Edit

```
TimeSpan duration = TimeSpan.FromHours(2);
```

---

Type	Size	Range (approximate)	Precision	Use Case Example
byte	1 byte	0 to 255	Integer	Small numbers (e.g., binary data)
sbyte	1 byte	-128 to 127	Integer	Signed small numbers
short	2 bytes	-32,768 to 32,767	Integer	Medium-sized integers
ushort	2 bytes	0 to 65,535	Integer	Unsigned medium numbers
int	4 bytes	-2,147,483,648 to 2,147,483,647	Integer	Most common integer type
uint	4 bytes	0 to 4,294,967,295	Integer	Unsigned large numbers
long	8 bytes	$\pm 9,223,372,036,854,775,807$	Integer	Very large integers
ulong	8 bytes	0 to 18,446,744,073,709,551,615	Integer	Unsigned very large numbers
float	4 bytes	$\sim \pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~7 digits	Fast math, lower precision
double	8 bytes	$\sim \pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-16 digits	Most common for decimals/fractions
decimal	16 bytes	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	28-29 digits	Financial calculations (high precision)



## How to Choose the Right One

### ◆ Use `int` :

- Most common integer type
- Default for counting, loops, indexing, etc.

### ◆ Use `float` or `double` :

- When you need decimals but **don't need high precision**
- Example: physics simulations, graphics

## ◆ Use decimal :

- High precision math — e.g., money, taxes
- Avoids floating-point rounding errors



## Examples

csharp

Copy Edit

```
int apples = 42;
float piApprox = 3.14f;
double pi = 3.14159265358979;
decimal money = 199.99m;
long starsInGalaxy = 100_000_000_000;
```



## Precision Warning Example

csharp

Copy Edit

```
Console.WriteLine(0.1 + 0.2); // Outputs 0.30000000000000004 (double imprecision)
```

csharp

Copy Edit

```
decimal a = 0.1m;
decimal b = 0.2m;
Console.WriteLine(a + b); // Outputs 0.3 (decimal precision)
```



## C# Access Modifiers Explained

Modifier	Accessible From...	Use Case Example
public	Anywhere (any class, any project that references it)	Libraries, APIs
private	Only within the same class	Encapsulation

Modifier	Accessible From...	Use Case Example
protected	Within the same class or a class that inherits it	Base class logic
internal	Within the same assembly (project)	Hides from other projects
protected internal	From same assembly OR from derived classes in any assembly	Hybrid use
private protected	From same class OR derived classes <b>in the same assembly</b> only	Most restrictive hybrid

---

## Examples of Each Modifier

### ◆ public

```
csharp Copy Edit  
  
public class Car  
{  
    public string Model;  
    public void Drive() => Console.WriteLine("Driving...");  
}
```

✓ Can be accessed anywhere.

---

### ◆ private *(default for class members)*

```
csharp Copy Edit  
  
public class Car  
{  
    private int speed;  
  
    private void Accelerate()  
    {  
        speed += 10;  
    }  
}
```



✗ Only accessible **inside the** `Car` class.

---

## ◆ `protected`

```
csharp Copy Edit

public class Vehicle
{
    protected int wheels = 4;
}

public class Bike : Vehicle
{
    public void ShowWheels() => Console.WriteLine(wheels); // ✓ Accessible
}
```

✓ Accessible in `Bike`, which inherits from `Vehicle`.

---

## ◆ `internal`

```
csharp Copy Edit

internal class SecretHelper
{
    internal void DoSomething() => Console.WriteLine("Hidden from outside assemblies");
}
```

✓ Accessible **only within the same project/assembly**.

---

## ◆ `protected internal`

```
cssharp Copy Edit

public class Animal
{
    protected internal void Breathe() => Console.WriteLine("Breathing...");
}
```

✓ Accessible in:

- Same assembly
- Derived classes, even from other assemblies

## ◆ private protected (C# 7.2+)

```
cssharp Copy Edit

public class Engine
{
    private protected void Start() => Console.WriteLine("Starting...");
}
```

✓ Accessible only to:

- The same class
- Derived classes **within the same assembly**

## 💡 Summary Table

Modifier	Same Class	Derived Class (Same Assembly)	Derived Class (Other Assembly)	Same Assembly
public	✓	✓	✓	✓
private	✓	✗	✗	✗
protected	✓	✓	✓	✗
internal	✓	✓	✗	✓

Modifier	Same Class	Derived Class (Same Assembly)	Derived Class (Other Assembly)	Same Assembly
protected internal	✓	✓	✓	✓
private protected	✓	✓	✗	✓

## What is Polymorphism?

**Polymorphism** = “many forms” — it allows one interface to behave differently depending on the type of object that implements or inherits it.

## Compile-time Polymorphism (Static Binding)

### ✓ Achieved Through:

- Method Overloading
- Operator Overloading

### Example:

```

csharp                                                                    Copy Edit
public class Calculator
{
    public int Add(int a, int b) => a + b;
    public double Add(double a, double b) => a + b;
}

```

✓ The correct `Add` method is chosen **at compile time** based on argument types.

### Key Traits:

- Resolved at **compile time**
- Faster
- Less flexible

- No inheritance required

---

## Runtime Polymorphism (Dynamic Binding)

### Achieved Through:

- **Method Overriding** using `virtual` and `override`
- **Interface implementation**

### Example:

```
cssharp                                                                    Copy Edit

public class Animal
{
    public virtual void Speak() => Console.WriteLine("Animal sound");
}

public class Dog : Animal
{
    public override void Speak() => Console.WriteLine("Bark");
}

public class Cat : Animal
{
    public override void Speak() => Console.WriteLine("Meow");
}

cssharp                                                                    Copy Edit

Animal pet = new Dog();
pet.Speak(); // Outputs: Bark
```

-  The correct `Speak` method is chosen **at runtime** based on the actual object type ( `Dog` ).

### Key Traits:

- Resolved at **runtime**
- Slower but more **flexible**
- Requires inheritance or interface

- Enables polymorphic behavior

---

## Comparison Table

Feature	Compile-time Polymorphism	Runtime Polymorphism
Binding Time	Compile time	Runtime
Achieved Using	Method Overloading / Operator Overloading	Method Overriding / Interfaces
Flexibility	Low	High
Speed	Faster	Slightly slower
Requires Inheritance?	❌ No	✅ Yes (or interface)
Example	Add(int, int) vs Add(double, double)	virtual Speak() / override Speak()

---

## Real-World Analogy

- **Compile-time polymorphism:** You choose between a hammer or a screwdriver **before** you start the job.
- **Runtime polymorphism:** You use a "tool" and based on the actual tool you're handed **during** the job, it behaves like a hammer or a screwdriver.

---

## 1. Factory Method

### Use Case:

Choose which subclass to instantiate **at runtime**, without exposing creation logic.

### Concept:

Define an interface for creating an object, but let subclasses decide which class to instantiate.

### Example:

```
// Product
public abstract class Animal
{
    public abstract string Speak();
}

// Concrete Products
public class Dog : Animal
{
    public override string Speak() => "Bark!";
}

public class Cat : Animal
{
    public override string Speak() => "Meow!";
}

// Creator
public abstract class AnimalFactory
{
    public abstract Animal CreateAnimal();
}

// Concrete Creators
public class DogFactory : AnimalFactory
{
    public override Animal CreateAnimal() => new Dog();
}

public class CatFactory : AnimalFactory
{
    public override Animal CreateAnimal() => new Cat();
}

// Usage
var factory = new DogFactory();
Animal pet = factory.CreateAnimal();
Console.WriteLine(pet.Speak()); // Outputs: Bark!
```



## 2. Abstract Factory

## Use Case:

Create **families of related objects** without specifying their concrete classes.

## Concept:

Each factory returns a group of related objects (products).

## Example:

```
// Abstract Products
public interface IButton { void Render(); }
public interface ICheckbox { void Render(); }

// Concrete Products
public class WinButton : IButton { public void Render() => Console.WriteLine("Windows Button"); }
public class MacButton : IButton { public void Render() => Console.WriteLine("Mac Button"); }

public class WinCheckbox : ICheckbox { public void Render() => Console.WriteLine("Windows Checkbox"); }
public class MacCheckbox : ICheckbox { public void Render() => Console.WriteLine("Mac Checkbox"); }

// Abstract Factory
public interface IGUIFactory
{
    IButton CreateButton();
    ICheckbox CreateCheckbox();
}

// Concrete Factories
public class WinFactory : IGUIFactory
{
    public IButton CreateButton() => new WinButton();
    public ICheckbox CreateCheckbox() => new WinCheckbox();
}
```

```
public class MacFactory : IGUIFactory
{
    public IButton CreateButton() => new MacButton();
    public ICheckbox CreateCheckbox() => new MacCheckbox();
}

// Usage
IGUIFactory factory = new MacFactory();
factory.CreateButton().Render(); // Mac Button
factory.CreateCheckbox().Render(); // Mac Checkbox
```

## 3. Builder

### Use Case:

Build **complex objects step-by-step** (e.g. meal, document, UI).



## Concept:

Separate construction of a complex object from its representation.

## Example:

```
csharp Copy Edit

public class Burger
{
    public string Bread, Meat, Cheese, Veggies;
    public override string ToString() => $"Burger with {Bread}, {Meat}, {Cheese}, {Veggies}";
}

// Builder
public class BurgerBuilder
{
    private Burger burger = new Burger();

    public BurgerBuilder AddBread(string bread) { burger.Bread = bread; return this; }
    public BurgerBuilder AddMeat(string meat) { burger.Meat = meat; return this; }
    public BurgerBuilder AddCheese(string cheese) { burger.Cheese = cheese; return this; }
    public BurgerBuilder AddVeggies(string veggies) { burger.Veggies = veggies; return this; }
    public Burger Build() => burger;
}

// Usage
var builder = new BurgerBuilder();
Burger myBurger = builder.AddBread("Brioche").AddMeat("Beef").AddCheese("Cheddar").Build();
Console.WriteLine(myBurger); // Burger with Brioche, Beef, Cheddar,
```

---

## 4. Singleton

### Use Case:

Ensure only **one instance** of a class exists globally (e.g., logging, config, database).

## Concept:

Control access to the single instance.

## Example:


```
public class Logger
{
    private static Logger instance;
    private static readonly object padlock = new object();

    private Logger() { }

    public static Logger Instance
    {
        get
        {
            lock (padlock)
            {
                if (instance == null)
                    instance = new Logger();
                return instance;
            }
        }
    }

    public void Log(string message) => Console.WriteLine($"[LOG]: {message}");
}

// Usage
Logger.Instance.Log("App started");
```



---

## 5. Prototype

### Use Case:

Avoid costly creation by cloning existing objects.

### Concept:

Use a prototype object and clone it to make new objects.

## Example:

```
cssharp Copy Edit

public abstract class Shape
{
    public int X, Y;
    public abstract Shape Clone();
}

public class Circle : Shape
{
    public int Radius;

    public override Shape Clone()
    {
        return new Circle { X = this.X, Y = this.Y, Radius = this.Radius };
    }
}

// Usage
var original = new Circle { X = 5, Y = 10, Radius = 20 };
var copy = original.Clone();
Console.WriteLine($"Original: {original.X}, Copy: {copy.X}");
```

## Summary Table

Pattern	Purpose	Key Concept
Factory Method	Delegates instantiation to subclasses	Subclass decides what to create
Abstract Factory	Create families of related objects	Group creation of objects
Builder	Step-by-step object construction	Separate building from object
Singleton	One instance globally	Controlled access, static prop
Prototype	Clone existing object	Copy instead of recreate

## 1. Adapter

### Use Case:

Make two incompatible interfaces work together.

## Concept:

Wrap an existing class with a new interface.

## Example:

```
cssharp Copy Edit

// Old system
public class LegacyPrinter
{
    public void PrintOld(string text) => Console.WriteLine("Legacy: " + text);
}

// New interface
public interface IPrinter
{
    void Print(string message);
}

// Adapter
public class PrinterAdapter : IPrinter
{
    private readonly LegacyPrinter _legacy;
    public PrinterAdapter(LegacyPrinter legacy) => _legacy = legacy;

    public void Print(string message) => _legacy.PrintOld(message);
}

// Usage
IPrinter printer = new PrinterAdapter(new LegacyPrinter());
printer.Print("Hello"); // Output: Legacy: He ↓
```

---

## 2. Decorator

### Use Case:

Add responsibilities to objects **dynamically** without modifying them.

## Concept:

Wrap the object and extend its behavior.

## Example:

```
public interface IMessage
{
    string GetContent();
}

// Base
public class SimpleMessage : IMessage
{
    public string GetContent() => "Hello";
}

// Decorator
public class EncryptedMessage : IMessage
{
    private readonly IMessage _message;
    public EncryptedMessage(IMessage message) => _message = message;

    public string GetContent() => Convert.ToBase64String(System.Text.Encoding.UTF8.GetBytes(_message.GetContent()));
}

// Usage
IMessage msg = new EncryptedMessage(new SimpleMessage());
Console.WriteLine(msg.GetContent()); // Outputs encrypted Hello
```

---

## 3. Facade

### Use Case:

Simplify access to a **complex subsystem**.

### Concept:

Provide a unified, higher-level interface.

## Example:

```
// Subsystems
public class CPU { public void Start() => Console.WriteLine("CPU started"); }
public class Memory { public void Load() => Console.WriteLine("Memory loaded"); }
public class Disk { public void Read() => Console.WriteLine("Disk read"); }

// Facade
public class Computer
{
    private CPU cpu = new CPU();
    private Memory memory = new Memory();
    private Disk disk = new Disk();

    public void Start()
    {
        cpu.Start();
        memory.Load();
        disk.Read();
    }
}

// Usage
var pc = new Computer();
pc.Start(); // Hides complexity
```

## 4. Proxy

### Use Case:

Control access to an object (e.g., lazy load, access control, logging).

### Concept:

A placeholder or surrogate for another object.

### Example:

```
public interface IService
{
    void Request();
}

// Real Subject
public class RealService : IService
{
    public void Request() => Console.WriteLine("Request handled.");
}

// Proxy
public class ServiceProxy : IService
{
    private RealService _realService;
    public void Request()
    {
        Console.WriteLine("Proxy: Logging before request...");
        _realService ??= new RealService();
        _realService.Request();
    }
}

// Usage
IService service = new ServiceProxy();
service.Request(); // Logs and calls real obje
```

Copy Edit

---

## 5. Flyweight

### Use Case:

Reduce memory by sharing common state between objects.

### Concept:

Store **intrinsic (shared)** state in flyweights, and **extrinsic** state outside.

### Example:

```
public class Character
{
    private readonly char _symbol;
    public Character(char symbol) => _symbol = symbol;

    public void Display(string font)
    {
        Console.WriteLine($"Displaying '{_symbol}' in font {font}");
    }
}

// Flyweight Factory
public class CharacterFactory
{
    private Dictionary<char, Character> _chars = new();

    public Character Get(char key)
    {
        if (!_chars.ContainsKey(key))
            _chars[key] = new Character(key);
        return _chars[key];
    }
}
```



```
// Usage
var factory = new CharacterFactory();
Character a1 = factory.Get('a');
Character a2 = factory.Get('a');
Console.WriteLine(object.ReferenceEquals(a1, a2)); // True - shared instance
```



## 6. Bridge



### Use Case:

Decouple abstraction from implementation — so both can evolve independently.





### Concept:

Split large class/interface into **two hierarchies**: abstraction and implementation.



## Example:

csharp

 Copy  Edit

```
// Implementation
public interface IRenderer
{
    void Render(string shape);
}

public class VectorRenderer : IRenderer
{
    public void Render(string shape) => Console.WriteLine($"Drawing {shape} as lines");
}

public class RasterRenderer : IRenderer
{
    public void Render(string shape) => Console.WriteLine($"Drawing {shape} as pixels");
}

// Abstraction
public class Shape
{
    protected IRenderer _renderer;
    public Shape(IRenderer renderer) => _renderer = renderer;

    public virtual void Draw() { }
}
```

```
// Refined Abstraction
public class Circle : Shape
{
    public Circle(IRenderer renderer) : base(renderer) { }
    public override void Draw() => _renderer.Render("Circle");
}

// Usage
Shape shape = new Circle(new VectorRenderer());
shape.Draw(); // Drawing Circle as lines
```

---

## Summary Table

Pattern	Purpose	Key Idea
<b>Adapter</b>	Connect incompatible interfaces	Wraps old interface to match new
<b>Decorator</b>	Add behavior dynamically	Wrap object to extend functionality
<b>Facade</b>	Simplify a complex system	Unified interface for subsystem
<b>Proxy</b>	Control access to another object	Stand-in for real object
<b>Flyweight</b>	Reduce memory use	Share common data
<b>Bridge</b>	Decouple abstraction from implementation	Separate abstraction from details



## 1. Command



### Use Case:

Encapsulate a **request as an object**, letting you parametrize and queue operations.



### Concept:

Separate the **sender** of a request from its **executor**.



### Example:

```
// Receiver
public class Light
{
    public void TurnOn() => Console.WriteLine("Light On");
    public void TurnOff() => Console.WriteLine("Light Off");
}

// Command interface
public interface ICommand
{
    void Execute();
}

// Concrete Commands
public class TurnOnCommand : ICommand
{
    private readonly Light _light;
    public TurnOnCommand(Light light) => _light = light;
    public void Execute() => _light.TurnOn();
}

// Invoker
public class RemoteControl
{
    public void Submit(ICommand command) => command.Execute();
}

// Usage
var light = new Light();
var turnOn = new TurnOnCommand(light);
var remote = new RemoteControl();
remote.Submit(turnOn); // Light On
```



## 2. Mediator



### Use Case:

Reduce complexity by centralizing communication **between objects**.

## **Concept:**

Instead of objects referencing each other directly, they communicate through a **mediator**.

## **Example:**

```
public interface IChatMediator
{
    void SendMessage(string message, User sender);
}

public class ChatRoom : IChatMediator
{
    private List<User> _users = new();

    public void Register(User user) => _users.Add(user);

    public void SendMessage(string message, User sender)
    {
        foreach (var user in _users.Where(u => u != sender))
            user.Receive(message);
    }
}
```

```
public class User
{
    public string Name { get; }
    private IChatMediator _mediator;

    public User(string name, IChatMediator mediator)
    {
        Name = name;
        _mediator = mediator;
        _mediator.Register(this);
    }

    public void Send(string msg) => _mediator.SendMessage(msg, this);
    public void Receive(string msg) => Console.WriteLine($"{Name} received: {msg}");
}

// Usage
var chat = new ChatRoom();
var john = new User("John", chat);
var anna = new User("Anna", chat);
john.Send("Hello!"); // Anna received: Hello!
```

## 3. Observer

### Use Case:



Notify **multiple objects** about state changes.

### Concept:

Subject holds a list of observers and notifies them automatically.

### Example:

csharp

 Copy  Edit

```
public interface IObserver
{
    void Update(string news);
}

public class NewsAgency
{
    private List<IObserver> observers = new();
    public void Register(IObserver o) => observers.Add(o);
    public void Notify(string news)
    {
        foreach (var o in observers)
            o.Update(news);
    }
}

public class NewsChannel : IObserver
{
    public string LatestNews { get; private set; }
    public void Update(string news) => LatestNews = news;
}
```

```
// Usage
var agency = new NewsAgency();
var channel = new NewsChannel();
agency.Register(channel);
agency.Notify("New president elected!");
Console.WriteLine(channel.LatestNews); // New president elected!
```



## 4. Strategy



### Use Case:

Choose an algorithm at runtime.



### Concept:

Define a family of algorithms, encapsulate each one, and make them interchangeable.



### Example:

```
public interface ISortStrategy
{
    void Sort(List<int> list);
}

public class QuickSort : ISortStrategy
{
    public void Sort(List<int> list) => Console.WriteLine("QuickSort executed");
}

public class BubbleSort : ISortStrategy
{
    public void Sort(List<int> list) => Console.WriteLine("BubbleSort executed");
}

public class Sorter
{
    private ISortStrategy _strategy;
    public Sorter(ISortStrategy strategy) => _strategy = strategy;
    public void Sort(List<int> list) => _strategy.Sort(list);
}

// Usage
var list = new List<int> { 5, 2, 9 };
var sorter = new Sorter(new QuickSort());
sorter.Sort(list); // QuickSort executed
```

[Copy](#) [Edit](#)

## 5. State



### Use Case:

Allow an object to change its **behavior when its state changes**.

### **Concept:**

Encapsulate states into separate classes and switch between them.

### **Example:**



```
public interface IState
{
    void Handle(Context context);
}

public class StartState : IState
{
    public void Handle(Context context)
    {
        Console.WriteLine("Starting...");
        context.State = new StopState();
    }
}

public class StopState : IState
{
    public void Handle(Context context)
    {
        Console.WriteLine("Stopping...");
        context.State = new StartState();
    }
}
```

```
public class Context
{
    public IState State { get; set; }

    public Context(IState state) => State = state;

    public void Request() => State.Handle(this);
}

// Usage
var context = new Context(new StartState());
context.Request(); // Starting...
context.Request(); // Stopping...
```



## 6. Memento

## ✓ Use Case:

Restore an object to its **previous state** (undo functionality).

## 🧠 Concept:

Save and restore state without violating encapsulation.

## 🔧 Example:

```
public class Memento
{
    public string State { get; }
    public Memento(string state) => State = state;
}

public class Originator
{
    public string State { get; set; }

    public Memento Save() => new Memento(State);
    public void Restore(Memento memento) => State = memento.State;
}

public class Caretaker
{
    public Memento Memento { get; set; }
}

// Usage
var originator = new Originator { State = "On" };
var caretaker = new Caretaker { Memento = originator.Save() };
originator.State = "Off";
originator.Restore(caretaker.Memento);
Console.WriteLine(originator.State); // On
```

## 📌 Summary Table

Pattern	Purpose	Key Concept
Command	Encapsulate requests as objects	Decouple sender from receiver

Pattern	Purpose	Key Concept
Mediator	Centralize communication	Avoid direct references between objects
Observer	Notify multiple objects	Event-like subscription
Strategy	Swap algorithms at runtime	Encapsulate logic variants
State	Change behavior with internal state	State objects handle transitions
Memento	Save and restore object state	Externalize internal state safely



## 1. StackPanel

Arranges children in a vertical or horizontal line.

xml

Copy Edit

```
<StackPanel Orientation="Vertical">
  <TextBlock Text="Item 1"/>
  <TextBlock Text="Item 2"/>
</StackPanel>
```



## 2. Grid

Arranges content in rows and columns (like a table).

xml

Copy Edit

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>

  <TextBlock Grid.Row="0" Grid.Column="0" Text="Left" />
  <TextBlock Grid.Row="0" Grid.Column="1" Text="Right" />
</Grid>
```



### 3. DockPanel

"Docks" children to top, bottom, left, right; last child fills the rest.

xml

Copy Edit

```
<DockPanel>
  <TextBlock DockPanel.Dock="Top" Text="Header" />
  <TextBlock DockPanel.Dock="Bottom" Text="Footer" />
  <TextBlock Text="Main Content" />
</DockPanel>
```



### 4. Canvas

Free positioning with absolute Top , Left .

xml

Copy Edit

```
<Canvas>
  <Button Canvas.Top="10" Canvas.Left="20" Content="Click Me" />
</Canvas>
```



## 5. WrapPanel

Wraps child elements into the next row or column when no space.

```
xml                                                                    Copy Edit

<WrapPanel>
  <Button Content="1"/>
  <Button Content="2"/>
  <Button Content="3"/>
</WrapPanel>
```



## 6. TextBlock

Displays read-only text.

```
xml                                                                    Copy Edit

<TextBlock Text="Hello, Avalonia!" FontSize="18"/>
```



## 7. TextBox

Editable single- or multi-line text input.

```
xml                                                                    Copy Edit

<TextBox Watermark="Enter text here..."/>
```



## 8. Button

Clickable UI element.

xml

Copy Edit

```
<Button Content="Click Me" Click="Button_Click"/>
```



## 9. CheckBox

Boolean toggle.

xml

Copy Edit

```
<CheckBox Content="I agree"/>
```



## 10. RadioButton

Mutually exclusive option within a group.

xml

Copy Edit

```
<StackPanel>
  <RadioButton Content="Option 1" GroupName="Group1"/>
  <RadioButton Content="Option 2" GroupName="Group1"/>
</StackPanel>
```



## 11. ComboBox

Dropdown menu.

xml

Copy Edit

```
<ComboBox>
  <ComboBoxItem Content="Item A"/>
  <ComboBoxItem Content="Item B"/>
</ComboBox>
```



## 12. ListBox

List of selectable items.

xml

Copy Edit

```
<ListBox>
  <ListBoxItem Content="Item 1"/>
  <ListBoxItem Content="Item 2"/>
</ListBox>
```



## 13. Image

Displays images from a file or resource.

xml

Copy Edit

```
<Image Source="avares://MyApp/Assets/logo.png" Width="100"/>
```



## 14. Border

Wraps content with border and optional corner radius.

xml

Copy Edit

```
<Border BorderBrush="Gray" BorderThickness="2" CornerRadius="5">
  <TextBlock Text="Inside border"/>
</Border>
```



## 15. UserControl

Reusable composite UI component.

xml

Copy Edit

```
<UserControl xmlns="...">
  <StackPanel>
    <TextBlock Text="Reusable Section"/>
  </StackPanel>
</UserControl>
```

---

## Bonus: ContentControl

Displays a single piece of content. Useful for templating.

xml

Copy Edit

```
<ContentControl Content="{Binding SelectedView}"/>
```



## 1. Read CSV using CsvHelper

- ✓ Use CsvHelper NuGet package

- 🔧 Install via terminal: `dotnet add package CsvHelper`



```
csharp Copy Edit

using CsvHelper;
using System.Globalization;
using System.IO;
using System.Linq;

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

// Reading
using var reader = new StreamReader("people.csv");
using var csv = new CsvReader(reader, CultureInfo.InvariantCulture);
var people = csv.GetRecords<Person>().ToList();
```

people.csv :

```
pgsql Copy Edit

Name, Age
Alice, 30
Bob, 25
```



## 2. Read JSON File

✓ Uses built-in `System.Text.Json`

```
csharp Copy Edit

using System.Text.Json;

var jsonString = File.ReadAllText("data.json");
var people = JsonSerializer.Deserialize<List<Person>>(jsonString);
```

data.json :

```
json

[
  { "Name": "Alice", "Age": 30 },
  { "Name": "Bob", "Age": 25 }
]
```

## 3. Read Plain Text File

- ✓ Line-by-line or full content

```
csharp

// All lines as list
var lines = File.ReadAllLines("notes.txt");

// Whole file as string
var content = File.ReadAllText("notes.txt");
```

 notes.txt :

```
scss

Line one
Line two
Line three
```

# LINQ Commands & Use Cases

## 1. Where

- **What:** Filters a sequence based on a predicate (condition).
- **Use case:** Select only elements that satisfy a condition.

csharp

Copy Edit

```
var adults = people.Where(p => p.Age >= 18);
```

---

## 2. Select

- **What:** Projects each element into a new form.
- **Use case:** Transform or map elements.

csharp

Copy Edit

```
var names = people.Select(p => p.Name);
```

---

## 3. OrderBy / OrderByDescending

- **What:** Sorts elements ascending or descending.
- **Use case:** Sort collection by a key.

csharp

Copy Edit

```
var sorted = people.OrderBy(p => p.Age);
```

---

## 4. GroupBy

- **What:** Groups elements by a key.
- **Use case:** Categorize items into groups.

csharp

Copy Edit

```
var grouped = people.GroupBy(p => p.Age);
```

---

## 5. Join

- **What:** Joins two sequences based on matching keys.
- **Use case:** Combine data from two collections.

csharp

Copy Edit

```
var joinResult = people.Join(departments,  
    p => p.DepartmentId,  
    d => d.Id,  
    (p, d) => new { p.Name, Department = d.Name });
```

---

## 6. Any

- **What:** Checks if any element satisfies a condition.
- **Use case:** Quick existence check.

csharp

Copy Edit

```
bool hasTeenager = people.Any(p => p.Age < 20);
```

---

## 7. All

- **What:** Checks if all elements satisfy a condition.
- **Use case:** Validate entire collection.

csharp

Copy Edit

```
bool allAdults = people.All(p => p.Age >= 18);
```

---

## 8. First / FirstOrDefault

- **What:** Returns first element (or default if none).

- **Use case:** Get first matching or safe null.

```
csharp Copy Edit  
  
var firstAdult = people.FirstOrDefault(p => p.Age >= 18);
```

---

## 9. Last / LastOrDefault

- **What:** Returns last element (or default).
- **Use case:** Similar to First but last element.

```
csharp Copy Edit  
  
var lastPerson = people.Last();
```

---

## 10. Count

- **What:** Counts elements, optionally filtered.
- **Use case:** Get total or conditional count.

```
csharp Copy Edit  
  
int adultsCount = people.Count(p => p.Age >= 18);
```

---

## 11. Sum / Average / Min / Max

- **What:** Aggregate numeric data.
- **Use case:** Calculate totals, averages, or extremes.

```
csharp Copy Edit  
  
var totalAge = people.Sum(p => p.Age);  
var averageAge = people.Average(p => p.Age);
```

---

## 12. Distinct

- **What:** Removes duplicates.
- **Use case:** Get unique elements.

```
csharp                                                                    Copy Edit

var uniqueAges = people.Select(p => p.Age).Distinct();
```

---

## 13. Skip / Take

- **What:** Skip some elements and take others.
- **Use case:** Paging or limiting results.

```
csharp                                                                    Copy Edit

var page = people.Skip(10).Take(5);
```

---

## 14. Concat / Union / Intersect / Except

- **What:** Combine or compare sequences.
- **Use case:** Merge or find differences.

```
csharp                                                                    Copy Edit

var allPeople = list1.Concat(list2);
var common = list1.Intersect(list2);
```

---

## Summary Table

Command	Purpose	Use Case
Where	Filter elements	Select items by condition
Select	Map or project elements	Transform data
OrderBy	Sort ascending	Sort by property
GroupBy	Group elements	Categorize
Join	Combine sequences	Relate two collections
Any	Check existence	Is there any matching element?
All	Check all satisfy condition	Validate all elements
First	Get first element	Find first match or default
Count	Count elements	Get number of items
Sum / Average	Aggregate numbers	Calculate totals or averages
Distinct	Remove duplicates	Unique elements
Skip / Take	Pagination	Paging results
Concat / Union	Combine collections	Merge or find common/different

---

# SOLID Principles in OOP

---

## 1. S — Single Responsibility Principle (SRP)

- **Definition:** A class should have **only one reason to change** — it should have **only one responsibility**.
- **Why:** Keeps code focused, easier to maintain, and reduces bugs.
- **Example:** A `User` class handles user data; a `UserRepository` handles database operations. Don't mix both.

---

## 2. O — Open/Closed Principle (OCP)

- **Definition:** Software entities (classes, modules, functions) should be **open for extension but closed for modification**.
- **Why:** Allows you to add new features without changing existing code, reducing risk of bugs.

- **Example:** Use inheritance or interfaces to add new behaviors instead of modifying existing classes.
- 

### 3. L — Liskov Substitution Principle (LSP)

- **Definition:** Objects of a base class should be replaceable with objects of derived classes **without breaking the correctness** of the program.
  - **Why:** Ensures subclasses behave like their parent class and clients can use them interchangeably.
  - **Example:** If `Rectangle` is a base class, `Square` subclass should honor the behavior expected from `Rectangle` (don't violate expectations).
- 

### 4. I — Interface Segregation Principle (ISP)

- **Definition:** Clients should **not be forced to depend on interfaces they do not use**. Split interfaces into smaller, specific ones.
  - **Why:** Avoids bloated interfaces and reduces impact of changes.
  - **Example:** Instead of one big `IMachine` interface with `Print()`, `Scan()`, and `Fax()`, create smaller interfaces like `IPrinter`, `IScanner`.
- 

### 5. D — Dependency Inversion Principle (DIP)

- **Definition:**
    - High-level modules should **not depend on low-level modules**. Both should depend on abstractions (interfaces or abstract classes).
    - Abstractions should not depend on details. Details should depend on abstractions.
  - **Why:** Decouples code, making it easier to swap implementations and test.
  - **Example:** Instead of a `ReportGenerator` directly creating a `SqlDatabase` object, it should depend on an interface `IDatabase`. You can inject any implementation.
- 

## Summary Table



Principle	Key Idea	Why it matters
<b>SRP</b>	One class, one responsibility	Easier to maintain and understand
<b>OCP</b>	Open to extension, closed to change	Add features safely
<b>LSP</b>	Subtypes replace base types safely	Reliable polymorphism
<b>ISP</b>	Small, specific interfaces	Avoid forcing clients to implement unused methods
<b>DIP</b>	Depend on abstractions, not concretions	Loosely coupled, flexible code

---

## 1 Create Avalonia Project with Community Toolkit

```
bash                                                                    Copy Edit

# Create a new Avalonia MVVM application (recommended template)
dotnet new avalonia.mvvm -o MyAvaloniaApp
cd MyAvaloniaApp

# Add CommunityToolkit.Mvvm package (MVVM helpers)
dotnet add package CommunityToolkit.Mvvm
```

---

## 2 Create Console Project

```
bash                                                                    Copy Edit

dotnet new console -o MyConsoleApp
cd MyConsoleApp
```

---

**Summary:**

Task	Command
Create Avalonia MVVM app	<code>dotnet new avalonia.mvvm -o MyAvaloniaApp</code>
Add CommunityToolkit.Mvvm	<code>dotnet add package CommunityToolkit.Mvvm</code>
Create console app	<code>dotnet new console -o MyConsoleApp</code>

---

To add **CsvHelper** to your C# project using the .NET CLI, just run:

```
bash
dotnet add package CsvHelper
```

This will install the latest CsvHelper NuGet package into your current project.

---

### Example:

If your project folder is `MyConsoleApp`, do:

```
bash
cd MyConsoleApp
dotnet add package CsvHelper
```

Then you can start using it in your code with:

```
csharp
using CsvHelper;
```

---

### Example: Map "Time Stamp" CSV column to Timestamp property

```
using CsvHelper;
using CsvHelper.Configuration;
using System.Globalization;
using System.IO;

public class Record
{
    public DateTime Timestamp { get; set; }
    public string Name { get; set; }
}

// Create a mapping class
public sealed class RecordMap : ClassMap<Record>
{
    public RecordMap()
    {
        Map(m => m.Timestamp).Name("Time Stamp"); // Map CSV "Time Stamp" to Timestamp
        Map(m => m.Name).Name("Name");             // Map CSV "Name" to Name property
    }
}

// Reading CSV with mapping
using var reader = new StreamReader("data.csv");
using var csv = new CsvReader(reader, CultureInfo.InvariantCulture);

// Register the mapping
csv.Context.RegisterClassMap<RecordMap>();

var records = csv.GetRecords<Record>().ToList();
```

---

## What happens here?

- The CSV file may have a header like:  
Time Stamp,Name
  - Your C# class property is called Timestamp, but the CSV column is "Time Stamp".
  - ClassMap tells CsvHelper to translate "Time Stamp" to the Timestamp property.
-

## ◆ 1. Thread : Manual Multithreading

csharp

Copy Edit

```
using System.Threading;

Thread t = new Thread(() =>
{
    Console.WriteLine("Running in a separate thread");
});
t.Start();
```

- **Use Case:** Lower-level threading control.
  - **Drawbacks:** More overhead, no return values, limited exception handling.
- 

## ◆ 2. Task : Modern Asynchronous Execution

csharp

Copy Edit

```
using System.Threading.Tasks;

Task.Run(() =>
{
    Console.WriteLine("Running in a task");
});
```

- **Use Case:** Fire-and-forget or return values.
  - Better managed than Thread .
- 

## ◆ 3. async / await : Asynchronous Programming

csharp

Copy Edit

```
public async Task DownloadAsync()
{
    var client = new HttpClient();
    string result = await client.GetStringAsync("https://example.com");
    Console.WriteLine(result);
}
```

- **Use Case:** Non-blocking IO operations.
  - **Key Benefit:** Frees up threads during IO (e.g., web requests, file access).
- 

## ◆ 4. Returning Values from Tasks

csharp

Copy Edit

```
public async Task<int> GetDataAsync()
{
    await Task.Delay(1000);
    return 42;
}
```

## ◆ 5. lock : Synchronization

csharp

Copy Edit

```
private static readonly object _lock = new object();
private static int counter = 0;

public static void Increment()
{
    lock (_lock)
    {
        counter++; // Only one thread can access this at a time
    }
}
```

- **Use Case:** Protect shared data from race conditions.
- Use `lock` around code that must be atomic/thread-safe.

---

## ◆ 6. Thread-Safe Collections (from `System.Collections.Concurrent` )

Collection	Description
<code>ConcurrentDictionary&lt;K, V&gt;</code>	Thread-safe key-value store
<code>ConcurrentBag&lt;T&gt;</code>	Unordered thread-safe list
<code>ConcurrentQueue&lt;T&gt;</code>	Thread-safe queue
<code>ConcurrentStack&lt;T&gt;</code>	Thread-safe stack

### Example:

```
csharpCopy Edit  
  
using System.Collections.Concurrent;  
  
var dict = new ConcurrentDictionary<string, int>();  
dict.TryAdd("apple", 1);  
dict.AddOrUpdate("apple", 1, (key, oldValue) => oldValue + 1);
```

- **Use Case:** Safely update or read from collections without needing manual `lock` .

---

## ◆ Summary Table

Feature	Use Case	Recommended When
Thread	Low-level control	Rare, for long-running threads
Task	Run async code	Most general-purpose cases
<code>async/await</code>	Async IO (network, file)	Modern, non-blocking APIs
<code>lock</code>	Shared resource protection	Avoid race conditions

Feature	Use Case	Recommended When
Concurrent*	Shared collection access	Multi-threaded apps