

Наследование, интерфейсы

1. Наследование

Наследование — это механизм, который позволяет создать новый класс (наследник) на основе уже существующего класса (базового). Наследник получает поля, свойства и методы базового класса и может:

- использовать унаследованное поведение "как есть",
- добавлять новое поведение и данные,
- изменять (переопределять) поведение базового класса.

Для чего это нужно:

- избегать дублирования кода (общая логика вынесена в базовый класс),
- логически группировать похожие сущности (например, `Vehicle` → `Car`, `Truck`, `Motorcycle`),
- строить иерархии типов для полиморфного использования (через базовый тип обращаться к любому наследнику).

В C# — **одиночное наследование классов**: класс может наследоваться только от одного базового класса, в отличие от некоторых других языков.

Все классы в C# явно или неявно наследуются от `object` — у каждого класса есть методы `ToString()`, `GetHashCode()`, `Equals()` и т.д., которые можно переопределить.

Простейшая иерархия (пример)

```
class Vehicle
{
    public string Model { get; set; }

    public void Start()
    {
        Console.WriteLine($"{Model} engine started.");
    }
}

class Car : Vehicle
{
    public int Seats { get; set; }
}

class Truck : Vehicle
{
    public double LoadCapacity { get; set; }
}
```

Использование:

```
var c = new Car { Model = "Toyota", Seats = 5 };
c.Start(); // "Toyota engine started."
```

Конструкторы и `base(...)`

Когда создаётся объект наследника, сначала выполняется конструктор базового класса, затем — конструктора наследника (вызов базового конструктора происходит автоматически, если не указан явно).

```
class Vehicle
{
    public string Model { get; }
    public Vehicle(string model) => Model = model;
}

class Car : Vehicle
{
    public int Seats { get; }
    public Car(string model, int seats) : base(model)
    {
        Seats = seats;
    }
}
```

Если базовый конструктор требует аргументы, наследник обязан вызвать `: base(...)` в своём конструкторе. В противном случае компилятор выдаст ошибку.

protected и private

- `private` — доступен только внутри класса, где объявлен.
- `protected` — доступен внутри класса **и в классах-наследниках**.

Пример:

```
class Vehicle
{
    protected int _speed; // доступен в наследниках
    private string _secret; // недоступен в наследниках
}

class Car : Vehicle
{
    public void SetSpeed(int s) => _speed = s; // OK
    // _secret = "x"; // ошибка компиляции
}
```

`protected` полезен, когда нужно дать подклассу доступ к внутреннему состоянию, но не открывать это состояние наружу (вне иерархии).

Виртуальные методы, `override`, `new` и полиморфизм

- `virtual` указывает, что метод **можно переопределить** в наследниках.
- `override` — переопределение виртуального метода.
- `new` — скрывание метода базового класса — **не** то же самое, что `override` (это скрывание, не полиморфизм).

Полиморфизм: вы можете хранить ссылку на базовый класс и вызывать метод, реализация которого определяется во время выполнения в зависимости от реального типа объекта.

```
class Vehicle
{
    public string Model { get; set; }
    public virtual void Move() => Console.WriteLine($"{Model} moves");
}

class Car : Vehicle
{
    public override void Move() => Console.WriteLine($"{Model} drives on the road");
}

class Boat : Vehicle
{
    public override void Move() => Console.WriteLine($"{Model} sails on water");
}

Vehicle v1 = new Car { Model = "Toyota" };
Vehicle v2 = new Boat { Model = "Sailor" };

v1.Move(); // "Toyota drives on the road"
v2.Move(); // "Sailor sails on water"
```

Если вместо `override` использовать `new`, то поведение будет отличаться при обращении через ссылку базового типа:

```
class Base
{
    public virtual void Foo() => Console.WriteLine("Base.Foo");
}

class Derived : Base
{
    public new void Foo() => Console.WriteLine("Derived.Foo (hiding)");
}

Base b = new Derived();
b.Foo(); // вызов Base.Foo, т.к. потомок "скрыл" метод, а не переопределил
```

Поэтому для полиморфизма нужно использовать `virtual` + `override`.

sealed

- `sealed class` — класс, от которого **нельзя** наследоваться.
- `sealed` можно также применять к **переопределённым методам**, чтобы запретить дальнейшее переопределение в классах-потомках.

Пример:

```
sealed class FinalCar { }

// class SpecialCar : FinalCar {} // Ошибка: наследование запрещено

class Vehicle
{
    public virtual void Drive() { Console.WriteLine("Drive"); }
}

class Car : Vehicle
{
    public sealed override void Drive() { Console.WriteLine("Car drive"); }
}

class SportCar : Car
{
    // public override void Drive() { } // Ошибка: нельзя переопределять, метод sealed
}
```

`sealed` часто используют для безопасности/оптимизации, когда хотят гарантировать неизменность поведения.

Переопределение ToString()

Переопределение ToString() даёт удобное строковое представление объекта:

```
class Vehicle
{
    public string Model { get; set; }
    public override string ToString() => $"Vehicle: {Model}";
}

class Car : Vehicle
{
    public int Seats { get; set; }
    public override string ToString() => $"Car: {Model}, seats={Seats}";
}
```

Console.WriteLine(obj) автоматически вызывает obj.ToString() .

Абстрактные классы

- `abstract class` — класс, для которого нельзя создать экземпляр (`new` не разрешён).
- Может содержать **абстрактные методы** (без реализации) — наследники обязаны их реализовать.
- Может также содержать обычные (реализованные) методы и поля.

Пример:

```
abstract class Vehicle
{
    public string Model { get; set; }
    public abstract void Refuel(); // обязан реализовать наследник

    public void ShowModel() => Console.WriteLine(Model);
}

class ElectricCar : Vehicle
{
    public override void Refuel() => Console.WriteLine("Plug in to charge battery");
}
```

Абстрактный класс удобно использовать, когда есть общий каркас поведения и части реализации, которые должны быть реализованы конкретными наследниками.

Upcast и downcast

- **Upcast:** приведение типа наследника к базовому типу. Всегда безопасно и происходит неявно.

```
Car car = new Car();  
Vehicle v = car; // upcast – безопасно
```

- **Downcast:** приведение базового типа к конкретному наследнику. Может быть небезопасно (в результате — `InvalidCastException`) если фактический объект другого типа.

```
Vehicle v = new Car();  
Car c = (Car)v; // явный даункаст – ОК в этом примере  
  
Vehicle v2 = new Vehicle();  
Car c2 = (Car)v2; // InvalidCastException – v2 не Car
```

Проверка типов и безопасные приёмы:

- `is` / pattern matching:

```
if (v is Car car)
{
    car.OpenTrunk();
}
```

- `as` — пытается привести и возвращает `null` при неудаче (работает только с ссылочными типами):

```
Car c = v as Car;
if (c != null) c.OpenTrunk();
```

- `typeof(T)` — возвращает `Type` для типа `T`. Используется редко:

```
if (v.GetType() == typeof(Car)) { ... } // строгое совпадение типа
```

Важно: приведение **не создаёт новый объект**, оно лишь рассматривает уже существующий объект с другой статической точкой зрения (скрывая/показывая поля и методы).

Поведенческие паттерны (примеры)

Шаблонный метод (Template Method) — абстрактный класс задаёт алгоритм, шаги могут быть реализованы в наследниках

Сценарий: процесс `StartTrip()` для транспортного средства имеет общий каркас:

1. Проверить системные условия,
2. Выполнить типичный запуск двигателя,
3. Совершить поездку,
4. Завершить поездку.

Шаблонный метод (`StartTrip`) фиксирует порядок шагов; детали шагов реализуют наследники. Это удобно, когда общий алгоритм стабилен, но детали различаются.

```
using System;

abstract class Vehicle
{
    public string Model { get; }
    protected Vehicle(string model) => Model = model;

    // "Шаблонный" метод – задаёт алгоритм
    public void StartTrip()
    {
        CheckSystems();
        StartEngine();
        Drive();
        StopEngine();
        Console.WriteLine($"{Model}: trip finished.");
    }

    protected virtual void CheckSystems()
    {
        Console.WriteLine($"{Model}: checking basic systems...");
    }

    protected abstract void StartEngine(); // шаг без реализации
    protected abstract void Drive();      // шаг без реализации

    protected virtual void StopEngine()
    {
        Console.WriteLine($"{Model}: engine stopped.");
    }
}
```

```
class PetrolCar : Vehicle
{
    public PetrolCar(string model) : base(model) { }

    protected override void StartEngine() =>
        Console.WriteLine($"{Model}: starting petrol engine (turn key)...");

    protected override void Drive() =>
        Console.WriteLine($"{Model}: driving on fuel...");
}

class ElectricCar : Vehicle
{
    public ElectricCar(string model) : base(model) { }

    protected override void StartEngine() =>
        Console.WriteLine($"{Model}: powering electric motor (press button)...");

    protected override void Drive() =>
        Console.WriteLine($"{Model}: driving on electricity...");

    protected override void CheckSystems() =>
        Console.WriteLine($"{Model}: checking battery and electronics...");
}
```

```
// Пример использования:
var v1 = new PetrolCar("Toyota");
var v2 = new ElectricCar("Tesla");

v1.StartTrip();
// Toyota: checking basic systems...
// Toyota: starting petrol engine (turn key)...
// Toyota: driving on fuel...
// Toyota: engine stopped.
// Toyota: trip finished.

v2.StartTrip();
// Tesla: checking battery and electronics...
// Tesla: powering electric motor (press button)...
// Tesla: driving on electricity...
// Tesla: engine stopped.
// Tesla: trip finished.
```

Паттерн **Стратегия (Strategy)** предлагает вынести изменяемый алгоритм в отдельные классы (стратегии) и при необходимости подставлять нужную стратегию в объект-контекст.

Идея: у нас есть автомобиль (`CarWithFuelStrategy`), но он может работать на разном топливе — бензин, электричество, газ и т.п. Вместо того, чтобы жёстко «зашивать» вид топлива внутрь класса, мы вынесем эту логику в отдельные стратегии.

```
// Базовый класс стратегии
abstract class FuelStrategy
{
    public abstract void Refuel(string model);
}

// Конкретные стратегии
class PetrolFuelStrategy : FuelStrategy
{
    public override void Refuel(string model) =>
        Console.WriteLine($"{model}: заправляем бензином на заправке");
}

class ElectricFuelStrategy : FuelStrategy
{
    public override void Refuel(string model) =>
        Console.WriteLine($"{model}: подключаем к зарядной станции");
}

class GasFuelStrategy : FuelStrategy
{
    public override void Refuel(string model) =>
        Console.WriteLine($"{model}: заправляем газом (метан/пропан)");
}
```



```
// Контекст – автомобиль, который использует стратегию топлива
class CarWithFuelStrategy
{
    public string Model { get; }
    private FuelStrategy _fuelStrategy;

    public CarWithFuelStrategy(string model, FuelStrategy fuelStrategy)
    {
        Model = model;
        _fuelStrategy = fuelStrategy;
    }

    public void SetFuelStrategy(FuelStrategy strategy) => _fuelStrategy = strategy;

    public void Refuel() => _fuelStrategy.Refuel(Model);
}

// Использование:
var tesla = new CarWithFuelStrategy("Tesla", new ElectricFuelStrategy());
tesla.Refuel(); // "Tesla: подключаем к зарядной станции"

var toyota = new CarWithFuelStrategy("Toyota", new PetrolFuelStrategy());
toyota.Refuel(); // "Toyota: заправляем бензином на заправке"

toyota.SetFuelStrategy(new GasFuelStrategy());
toyota.Refuel(); // "Toyota: заправляем газом (метан/пропан)"
```

2. Интерфейсы

Интерфейс — это контракт, который описывает, *что умеет объект делать*, но не содержит реализации.

- В интерфейсе есть только **сигнатуры методов, свойств, событий**.
- Класс, реализующий интерфейс, обязан реализовать все его члены.
- Один класс может реализовывать **несколько интерфейсов** (в отличие от наследования, где можно наследоваться только от одного класса).

Это позволяет описывать общее поведение разных, несвязанных между собой объектов.

Синтаксис

```
interface IFlyable
{
    void Fly();
}

class Bird : IFlyable
{
    public void Fly() => Console.WriteLine("Птица машет крыльями и летит");
}
```

Правила именования:

- Названия интерфейсов в C# начинаются с буквы **I**: `Comparable` , `Disposable` , `IFlyable` .
- Это позволяет сразу отличить интерфейс от класса.

Интерфейсы vs Наследование

- **Наследование** связывает классы в иерархию.
- **Интерфейс** описывает поведение, которым может обладать совершенно разные классы (даже не связанные иерархией).

Например:

- `Bird` и `Airplane` никак не связаны по иерархии, но оба могут реализовать интерфейс `IFlyable`.

```
interface IFlyable
{
    void Fly();
}

class Bird : IFlyable
{
    public void Fly() => Console.WriteLine("Птица машет крыльями и летит");
}

class Airplane : IFlyable
{
    public void Fly() => Console.WriteLine("Самолет разгоняется и взлетает");
}
```

Пример «Стратегия» через интерфейсы

Перепишем пример с **топливом** на интерфейсы (это чаще встречается в C#):

```
interface IFuelStrategy
{
    void Refuel(string model);
}

class PetrolFuelStrategy : IFuelStrategy
{
    public void Refuel(string model) =>
        Console.WriteLine($"{model}: заправляем бензином");
}

class ElectricFuelStrategy : IFuelStrategy
{
    public void Refuel(string model) =>
        Console.WriteLine($"{model}: подключаем к зарядной станции");
}

class GasFuelStrategy : IFuelStrategy
{
    public void Refuel(string model) =>
        Console.WriteLine($"{model}: заправляем газом");
}
```

```
class Car
{
    public string Model { get; }
    private IFuelStrategy _fuelStrategy;

    public Car(string model, IFuelStrategy fuelStrategy)
    {
        Model = model;
        _fuelStrategy = fuelStrategy;
    }

    public void SetFuelStrategy(IFuelStrategy strategy) => _fuelStrategy = strategy;
    public void Refuel() => _fuelStrategy.Refuel(Model);
}

// Использование
var tesla = new Car("Tesla", new ElectricFuelStrategy());
tesla.Refuel(); // Tesla: подключаем к зарядной станции

var lada = new Car("Lada", new PetrolFuelStrategy());
lada.Refuel(); // Lada: заправляем бензином
```

Отличие абстрактного класса от интерфейса

Абстрактный класс	Интерфейс
Может содержать поля, свойства, реализацию методов	Не содержит полей, только сигнатуры
Наследование только от одного класса	Можно реализовать несколько интерфейсов
Используется, когда есть общая логика для группы классов	Используется, когда классы должны иметь одинаковое поведение, но не обязаны быть связаны

Паттерн «Команда» (Command)

Паттерн **Команда (Command)** позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

Это позволяет:

- отделить инициатора действия от получателя,
- хранить, передавать и комбинировать команды как объекты,
- легко добавлять новые команды, не изменяя старый код.

Пример: Телевизор и Пульт

Интерфейс команды

```
interface ICommand
{
    void Execute(); // выполнить команду
    void Undo();    // отменить команду
}
```

Получатель (Receiver) — Телевизор

```
class TV
{
    public void On() => Console.WriteLine("Телевизор включён");
    public void Off() => Console.WriteLine("Телевизор выключен");
    public void VolumeUp() => Console.WriteLine("Громкость увеличена");
    public void VolumeDown() => Console.WriteLine("Громкость уменьшена");
}
```

Конкретные команды

```
class TurnOnCommand : ICommand
{
    private TV _tv;
    public TurnOnCommand(TV tv) { _tv = tv; }
    public void Execute() => _tv.On();
    public void Undo() => _tv.Off();
}

class TurnOffCommand : ICommand
{
    private TV _tv;
    public TurnOffCommand(TV tv) { _tv = tv; }
    public void Execute() => _tv.Off();
    public void Undo() => _tv.On();
}

class VolumeUpCommand : ICommand
{
    private TV _tv;
    public VolumeUpCommand(TV tv) { _tv = tv; }
    public void Execute() => _tv.VolumeUp();
    public void Undo() => _tv.VolumeDown();
}
```


Инициатор (Invoker) — Пульт

```
class RemoteControl
{
    private ICommand _command;

    public void SetCommand(ICommand command) => _command = command;

    public void PressButton() => _command?.Execute();
    public void PressUndo() => _command?.Undo();
}
```

Использование

```
var tv = new TV();
var remote = new RemoteControl();

// Включить телевизор
remote.SetCommand(new TurnOnCommand(tv));
remote.PressButton(); // Телевизор включён
remote.PressUndo();   // Телевизор выключен

// Увеличить громкость
remote.SetCommand(new VolumeUpCommand(tv));
remote.PressButton(); // Громкость увеличена
remote.PressUndo();   // Громкость уменьшена
```

Ссылки

- [Учебник по языку C# 13 и платформе .NET 9](#)
 - [C# и .NET | Архитектура приложений и паттерны проектирования](#)
 - Книга: Паттерны проектирования на платформе .NET, Тепляков С.
-

Практическое задание

Что нужно сделать:

1. Продолжайте работу над проектом `ToDoList`.
2. Создайте интерфейс `ICommand` :
 - Определите единственный метод `void Execute()` :
3. Создайте классы конкретных команд, которые реализуют `ICommand` :

- `AddCommand` , `ViewCommand` и т.п.

Требования к классам команд:

- Флаги команд оформите как свойства `bool` .
- Введённый текст или индекс задачи храните в свойствах типа `string` или `int` .
- Добавьте свойства `ToDoList` и/или `Profile` в зависимости от того, с чем работает команда.
- В методе `Execute()` вызываются **соответствующие методы** работы с `ToDoList` или `Profile` .

4. Создайте статический класс `CommandParser` :

- Метод:

```
csharp public static ICommand Parse(string inputString, ToDoList todoList, Profile profile)
```

- Метод должен:

- Определять команду по введённой строке.
- Разбирать флаги и параметры.
- Создавать объект соответствующего класса команды.
- Присваивать объекту свойства `ToDoList` или `Profile` в зависимости от назначения команды.
- Возвращать объект команды как `ICommand` .

5. Главный цикл программы (Main) теперь должен работать так:

- Считывать строку пользователя.
- Передавать её в `CommandParser.Parse()` вместе с текущими `TodoList` и `Profile`.
- Получать объект `ICommand`.
- Вызывать `Execute()` для выполнения команды.

Пример:

```
string input = Console.ReadLine();  
ICommand command = CommandParser.Parse(input, todoList, profile);  
command.Execute();
```

6. Тестирование программы:

- Проверить добавление задач (`add` и `add --multiline`).
- Просмотр задач с разными флагами (`view`).
- Отметка задач выполненными, обновление, удаление.
- Работа с командами `read` и `profile`.

7. Делайте коммиты после **каждого изменения**. Один большой коммит будет оцениваться в два раза ниже.

8. Обновите **README.md** — добавьте описание новых возможностей программы.

9. Сделайте push изменений в GitHub.