Делегаты и события

1. Что такое делегаты

В предыдущих темах мы работали в основном с **классами** и **структурами**.
Классы и структуры описывают **данные** — то, *что* хранится в программе: числа, строки, объекты, коллекции.
Однако в программировании нам часто нужно описывать не только *данные*, но и **поведение** — *что нужно сделать* с этими данными. Для этого в C# существует специальный тип — **делегаты**.

Делегат — это тип, который **хранит ссылку на метод**.
Иными словами, это “переменная, в которой можно хранить функцию”.
При помощи делегатов мы можем **передавать методы как параметры**, вызывать их в нужный момент и даже менять “логику поведения” программы на лету.

Пример простого делегата

Допустим, у нас есть два метода, выполняющих разные операции над числами:

```
int Add(int a, int b) => a + b;
int Multiply(int a, int b) => a * b;
```

Теперь объявим делегат, который может хранить ссылку на любой метод, принимающий два `int` и возвращающий `int`:

```
public delegate int Operation(int x, int y);
```

Создадим переменную типа `Operation` и “передадим” в неё метод:

```
Operation op = Add;        // делегат указывает на метод Add
Console.WriteLine(op(3, 5)); // 8

op = Multiply;             // теперь делегат указывает на Multiply
Console.WriteLine(op(3, 5)); // 15
```

Здесь переменная `op` фактически **ссылается на метод**, и мы можем вызывать его так же, как обычную функцию.

Делегаты как способ делегирования действий

Название “делегат” произошло от английского *delegate* — “передавать полномочия”.
Именно это они и делают: **передают выполнение метода другому объекту или методу**.

Например, вы можете написать метод, который выполняет какую-то задачу, но не знает *точно*, что именно нужно сделать с данными — он просто вызывает переданный делегат.

```
void ProcessNumbers(int a, int b, Operation operation)
{
    Console.WriteLine(operation(a, b));
}

// можно передавать разные методы:
ProcessNumbers(2, 3, Add);       // 5
ProcessNumbers(2, 3, Multiply);  // 6
```

Таким образом, делегаты позволяют **делать код гибким и расширяемым**:
один и тот же метод `ProcessNumbers` может работать с разной логикой, не изменяя свой код.

2. Как устроены делегаты

Мы уже знаем, что делегат — это переменная, которая может хранить ссылку на метод.
Но под капотом делегат — это **объект**, у которого есть два ключевых свойства:

- `Method` — ссылка на сам метод, который нужно вызвать.
- `Target` — объект, для которого этот метод должен быть вызван.

Таким образом, делегат можно представить как “обёртку” вокруг метода, в которой хранится информация: что вызывать и у кого вызывать.

- **Если метод статический**, у него нет привязки к конкретному объекту.
  В этом случае `Target` у делегата будет `null`, а `Method` просто указывает на статическую функцию.
- **Если метод динамический**, делегат хранит:
  - ссылку на сам метод (`Method`);
  - и ссылку на объект (`Target`), которому этот метод принадлежит.

Посмотрим на примере:

```
public delegate void ShowMessage();

class Greeter
{
    public void Hello() => Console.WriteLine("Hello!");
    public static void Bye() => Console.WriteLine("Goodbye!");
}

class Program
{
    static void Main()
    {
```

```
        var g = new Greeter();

        ShowMessage d1 = g.Hello;   // динамический метод
        ShowMessage d2 = Greeter.Bye; // статический метод

        Console.WriteLine($"d1.Target = {d1.Target}");
        Console.WriteLine($"d1.Method = {d1.Method}");

        Console.WriteLine($"d2.Target = {d2.Target}");
        Console.WriteLine($"d2.Method = {d2.Method}");

        d1(); // вызов Hello
        d2(); // вызов Bye
    }
}
```

Результат будет такой:

```
d1.Target = Greeter
d1.Method = Void Hello()
d2.Target =
d2.Method = Void Bye()
Hello!
Goodbye!
```

Делегат как параметр

Делегаты особенно полезны, когда нужно передать поведение в метод.
Например, метод может выполнять какую-то общую логику, но действие внутри будет определяться переданным делегатом:

```
public delegate void ActionDelegate(string text);

class Processor
{
    public void Process(string message, ActionDelegate action)
    {
        Console.WriteLine("Начало обработки...");
        action(message);
        Console.WriteLine("Обработка завершена.");
    }
}

class Program
{
    static void Main()
    {
        var p = new Processor();

        void PrintUpper(string text) => Console.WriteLine(text.ToUpper());
        void PrintLower(string text) => Console.WriteLine(text.ToLower());

        p.Process("Delegates are powerful!", PrintUpper);
        p.Process("Delegates are powerful!", PrintLower);
    }
}
```

Результат:

```
Начало обработки...
DELEGATES ARE POWERFUL!
Обработка завершена.

Начало обработки...
delegates are powerful!
Обработка завершена.
```

Здесь `Process` ничего не знает о том, **что именно** делать с текстом — он просто вызывает переданный ему делегат.

Таким образом, делегаты позволяют передавать “кусок поведения” в виде ссылки на метод,
а вызывающий код решает, *что именно* будет выполнено.

3. Обобщённые делегаты

Делегаты — это мощный инструмент, но создавать отдельный тип делегата для каждого метода неудобно.
Например, если нам нужно передать метод, который принимает `int` и возвращает `string`,
мы могли бы написать свой тип делегата:

```
public delegate string ConvertToString(int value);
```

Но таких комбинаций типов может быть очень много, поэтому в .NET есть **обобщённые (generic) делегаты** — готовые универсальные шаблоны для большинства задач.

Action

Используется, когда метод **ничего не возвращает** (`void`),
но принимает **один или несколько параметров**.

```
Action<string> print = text => Console.WriteLine($"Печатаем: {text}");
print("Привет, мир!");
```

Также существуют версии `Action` с несколькими параметрами:
`Action<T1, T2>`, `Action<T1, T2, T3>`, и т. д. (до 16 параметров).

```
Action<string, int> repeat = (word, count) =>
{
    for (int i = 0; i < count; i++)
        Console.Write(word + " ");
    Console.WriteLine();
};

repeat("Hello", 3); // Hello Hello Hello
```

Func<T, TResult>

Используется, когда метод **возвращает значение**.
Последний параметр — это тип возвращаемого значения (`TResult`).

```
Func<int, int, int> add = (a, b) => a + b;
int sum = add(3, 4);
Console.WriteLine(sum); // 7
```

Можно использовать несколько входных параметров:

```
Func<double, double, double, double> average = (a, b, c) => (a + b + c) / 3;
Console.WriteLine(average(3, 6, 9)); // 6
```

Predicate

Это частный случай `Func<T, bool>` — метод, который принимает один параметр и **возвращает логическое значение**.

Используется, например, для проверки условий.

```
Predicate<int> isEven = x => x % 2 == 0;

Console.WriteLine(isEven(4)); // True
Console.WriteLine(isEven(7)); // False
```

4. Анонимные делегаты и лямбда-выражения

Когда мы впервые сталкиваемся с делегатами, часто возникает проблема: нужно передать метод, но писать для этого **отдельный именованный метод** — слишком громоздко.

Пример:

```
delegate int Operation(int a, int b);

class Program
{
    static int Add(int a, int b) => a + b;

    static void Main()
    {
        Operation op = Add; // Работает, но приходится писать метод Add
        Console.WriteLine(op(3, 4)); // 7
    }
}
```

Решение: анонимные методы

В C# появилась возможность объявлять метод **прямо на месте**, без имени — с помощью ключевого слова `delegate`.

```
Operation op = delegate (int a, int b)
{
    return a + b;
};

Console.WriteLine(op(5, 6)); // 11
```

Лямбда-выражения

Позже в C# добавили более **удобный и лаконичный** синтаксис — **лямбда-выражения**.
Это короткая запись анонимных методов через стрелку `=>`.

```
Operation op = (a, b) => a + b;
Console.WriteLine(op(2, 3)); // 5
```

Синтаксис лямбда-выражений

Форма записи	Пример	Описание
Полная форма	`(int x, int y) => { return x + y; }`	Можно указывать типы и тело с `return`
Короткая форма	`(x, y) => x + y`	Типы выводятся автоматически
Одна переменная	`x => x * x`	Скобки можно опустить, если один параметр
Без параметров	`() => Console.WriteLine("Hello!")`	Используется, если метод ничего не принимает

Примеры:

```
Func<int, int> square = x => x * x;
Action sayHi = () => Console.WriteLine("Hello!");
Func<int, int, bool> isGreater = (a, b) => a > b;

Console.WriteLine(square(5));     // 25
sayHi();                          // Hello!
Console.WriteLine(isGreater(7,3)); // True
```

5. Замыкания

Замыкание — это механизм, при котором функция (например, лямбда) **захватывает** переменные из внешнего контекста, то есть получает доступ к локальным переменным, даже после выхода из метода, где они были объявлены.

Посмотрим простой пример:

```
Func<int, int> CreateAdder(int n)
{
    // n — это переменная внешнего контекста
    return x => x + n; // Лямбда замыкает переменную n
}

var add5 = CreateAdder(5);
var add10 = CreateAdder(10);

Console.WriteLine(add5(2));  // 7
Console.WriteLine(add10(2)); // 12
```

Здесь каждая лямбда “помнит” своё значение `n`.
Хотя метод `CreateAdder` уже завершил выполнение, переменная `n` всё ещё существует,
потому что её сохранило лямбда-выражение.

Пример с циклом и общей переменной

Замыкания часто приводят к неожиданным результатам, особенно в циклах. Например, предположим, что мы создаём список действий, которые должны выводить значение переменной `i`:

```
var actions = new List<Action>();

for (int i = 0; i < 3; i++)
{
    actions.Add(() => Console.WriteLine(i));
}

foreach (var action in actions)
    action();
```

Ожидаемый результат:

```
0
1
2
```

Реальный результат:

```
3
3
3
```

Так произошло потому что **все лямбды получили одну и ту же переменную** `i`,
а к моменту выполнения цикла `i` уже стала равна 3.

Как исправить

Нужно создать **новую локальную переменную внутри цикла**,
чтобы каждая лямбда замкнула свою собственную копию значения:

```
var actions = new List<Action>();

for (int i = 0; i < 3; i++)
{
    int local = i; // создаём копию
    actions.Add(() => Console.WriteLine(local));
}

foreach (var action in actions)
    action();
```

Теперь результат будет ожидаемым:

```
0
1
2
```

6. События

В реальных программах часто нужно, чтобы один объект мог **уведомлять другие объекты** о том, что произошло какое-то событие: нажатие кнопки, завершение загрузки, изменение значения и т. д. Для этого в C# существуют **события (events)**.

Событие — это механизм уведомления, позволяющий одному объекту (источнику события) оповещать других объектов (подписчиков) о том, что *“что-то произошло”*.

Вызывать событие (`Invoke`) может **только сам владелец события**, а подписываться (`+=`) и отписываться (`-=`) могут внешние объекты.

Объявление события

Чтобы объявить событие, используется ключевое слово `event`, после которого указывается тип делегата.

```
public event Action OnClick;
```

Это означает: “у этого объекта есть событие `OnClick`, на которое можно подписаться с помощью метода без параметров и возвращаемого значения”.

Подписка и отписка

Добавить подписчика можно с помощью оператора `+=`, а удалить — с помощью `-=`:

```
var button = new Button();

button.OnClick += () => Console.WriteLine("Кнопка нажата!");
button.OnClick += () => Console.WriteLine("Второй обработчик!");
button.Click(); // вызывает оба обработчика
button.OnClick -= () => Console.WriteLine("Кнопка нажата!"); // ⚠ не сработает — разные экземпляры лямбды
```

Чтобы корректно отписаться, нужно хранить ссылку на обработчик в переменной.

Пример: простая кнопка с событием `Clicked`

Создадим класс `Button`, который будет вызывать событие при нажатии:

```
class Button
{
    // Событие, на которое можно подписаться
    public event Action Clicked;
    // Метод, вызывающий событие
    public void Click()
    {
        Console.WriteLine("Нажата кнопка...");
        // Проверяем, есть ли подписчики
        Clicked?.Invoke(); // безопасный вызов (если Clicked != null)
    }
}
```

Использование:

```
class Program
{
    static void Main()
    {
        var button = new Button();

        // Подписываемся на событие
        button.Clicked += () => Console.WriteLine("Обработчик 1: пользователь нажал кнопку");
        button.Clicked += () => Console.WriteLine("Обработчик 2: сохраняем данные...");

        // Симулируем клик
        button.Click();
    }
}
```

Результат:

```
Нажата кнопка...
Обработчик 1: пользователь нажал кнопку
Обработчик 2: сохраняем данные...
```

Как это работает

- Код **внутри класса** может вызывать событие (`Invoke()`).
- Код **снаружи** может только **подписываться** (`+=`) или **отписываться** (`-=`).
- Событие хранит **список всех подписанных методов**.

Особенности событий

- Можно подписывать несколько обработчиков — события поддерживают **мультикаст** (вызов нескольких методов подряд).
- Можно использовать **лямбды**, **анонимные методы** или обычные именованные функции.
- Вызывать (`Invoke`) событие может **только владелец**, что защищает от случайных вызовов извне.
- Часто события используются в графических интерфейсах, сетевом коде и игровых движках.

Пример с параметрами события

Если нужно передавать данные при вызове события — можно использовать `Action<T>`.

```
class Downloader
{
    public event Action<int> ProgressChanged;
    public void Download()
    {
        for (int i = 0; i <= 100; i += 25)
            ProgressChanged?.Invoke(i);
    }
}
class Program
{
    static void Main()
    {
        var downloader = new Downloader();
        downloader.ProgressChanged += p => Console.WriteLine($"Загрузка: {p}%");
        downloader.Download();
    }
}
```

Результат:

```
Загрузка: 0%
Загрузка: 25%
Загрузка: 50%
Загрузка: 75%
Загрузка: 100%
```

8. Ковариация, контравариация и инвариантность

При работе с **обобщёнными типами** и **делегатами** часто возникает вопрос:
можно ли использовать объект *другого, производного или базового типа*?

Чтобы ответить на этот вопрос, нужно разобраться с **инвариантностью**, **ковариацией** и **контравариацией**.

Инвариантность

Инвариантность — это поведение “тип должен совпадать *точно*”.

Например, если метод принимает `List<object>`, то передать туда `List<string>` **нельзя**, даже несмотря на то, что `string` наследуется от `object`.

```
List<string> strings = new List<string> { "a", "b", "c" };
List<object> objects = strings; // ❌ Ошибка: несоответствие типов
```

Почему?
Потому что `List<T>` допускает изменение содержимого.
Если бы это было разрешено, можно было бы добавить в `objects` целое число,
а это нарушило бы типизацию исходного списка строк.

Ковариация (`out`)

Ковариация позволяет использовать **более конкретный тип**, когда ожидается **более общий**.

Например, если у нас есть коллекция `IEnumerable<string>`, мы можем передать её туда, где ожидается `IEnumerable<object>`.
Это логично, ведь строки — это объекты, и мы только **читаем** данные из коллекции.

```
IEnumerable<string> strings = new List<string> { "a", "b", "c" };
IEnumerable<object> objects = strings; // ✅ Разрешено (ковариация)
```

Как это работает?
Интерфейс `IEnumerable<T>` объявлен так:

```
public interface IEnumerable<out T> { ... }
```

Ключевое слово `out` указывает компилятору, что этот тип используется только как *результат* (возвращаемое значение), поэтому безопасно разрешить замену `T` на его базовый тип”.

Контравариация (`in`)

Контравариация — противоположная ситуация. Она позволяет использовать **более общий тип**, когда ожидается **более конкретный**.

Например, интерфейс `IComparer<T>` используется для сравнения элементов, и принимает значения типа `T` (входные параметры).

Он объявлен как:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Благодаря `in` , мы можем сделать следующее:

```
IComparer<object> objComparer = new MyObjectComparer();
IComparer<string> strComparer = objComparer; // ✅ Контравариация
```

Почему это безопасно?
Потому что метод сравнения принимает объекты (`object`), и он сможет сравнить и строки — ведь они тоже объекты.

Пример с делегатами

Делегаты тоже поддерживают ковариацию и контравариацию.

Ковариация — если метод возвращает **более конкретный тип**,
Контравариация — если метод принимает **более общий тип**.

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

delegate Animal CreateAnimal();
delegate void ProcessAnimal(Dog dog);

CreateAnimal creator = () => new Dog(); // ✅ Ковариация — возвращаем Dog вместо Animal

ProcessAnimal process = (Animal a) => Console.WriteLine(a.GetType().Name);
// ✅ Контравариация — принимаем Animal вместо Dog
```

Когда и зачем это нужно

Ковариация и контравариация позволяют писать более **гибкие** и **безопасные** обобщённые типы.
Они особенно полезны:

- при работе с интерфейсами (`IEnumerable<out T>` , `IComparer<in T>`);
- при создании собственных обобщённых делегатов;
- при реализации фабрик объектов, обработчиков событий, фильтров и т. д.

Главное правило:

Вид	Ключевое слово	Используется где	Принцип	Пример
Инвариантность	—	Точный тип	точь в точь	`List<T>`
Ковариация	`out`	Возвращаемое значение	текущий тип и типы наследники	`IEnumerable<out T>`
Контравариация	`in`	Входные параметры	текущий тип и родительские типы	`IComparer<in T>`

Практическое задание

Что нужно сделать:

Часть 1

Избавимся от множества `if` или `switch` в парсере с помощью делегатов..

1. **Добавить делегаты-обработчики команд:**
   - Создать в `CommandParser` словарь, где ключ — это название команды, а значение — делегат, который возвращает объект `ICommand`:
   `csharp private static Dictionary<string, Func<string, ICommand>> _commandHandlers = new();`
   - Формат делегата:
   `csharp Func<string, ICommand>`
   где входная строка — аргументы команды, а возвращаемое значение — готовый объект `ICommand`.
2. **Регистрировать обработчики для команд:**
   В методе `Init()` (или статическом конструкторе) зарегистрировать все команды:

```
_commandHandlers["add"] = ParseAdd;
_commandHandlers["delete"] = ParseDelete;
_commandHandlers["update"] = ParseUpdate;
...
```

3. **Изменить метод `Parse`:**
   Метод должен:
   - выделять из введённой строки название команды и аргументы;
   - получать обработчик из словаря по названию команды;
   - запускать его с переданными аргументами;
   - возвращать созданный объект `ICommand`.
     Пример реализации:

```
public static ICommand Parse(string input)
{
    var parts = input.Split(' ', 2);
    var command = parts[0].ToLower();
    var args = parts.Length > 1 ? parts[1] : "";

    if (_commandHandlers.TryGetValue(cmdName, out var handler))
        return handler(args);

    Console.WriteLine("Неизвестная команда.");
    return new HelpCommand();
}
```

Часть 2

В текущей реализации класс `TodoList` напрямую использует `FileManager` для сохранения и загрузки данных.

Такой подход затрудняет расширение системы (например, если в будущем данные нужно будет хранить не в файле, а в базе данных или на удалённом сервере).

Поэтому необходимо разделить их, реализовав взаимодействие через **события**.

1. **Добавить события в класс `TodoList`:**
   В классе `TodoList` определить события, которые будут вызываться при изменении данных:

   ```
   public event Action<TodoItem>? OnTodoAdded;
   public event Action<TodoItem>? OnTodoDeleted;
   public event Action<TodoItem>? OnTodoUpdated;
   public event Action<TodoItem>? OnStatusChanged;
   ```

   Эти события должны вызываться (через `?.Invoke()`) в соответствующих методах:
   - при добавлении задачи (`Add`);

- при удалении (`Delete`);
- при обновлении (`Update`);
- при изменении статуса (`SetStatus`);

2. **Удалить прямые вызовы `FileManager` из `TodoList`:**
   - Вместо этого в методах этих классов вызывать события:
     `csharp OnTaskAdded?.Invoke(item);`
     Таким образом, класс `TodoList` **не знает**, как и куда сохраняются данные — они просто сообщают, что данные изменились.
3. **Подписать `FileManager` на эти события:**
   После создания экземпляров `TodoList` в основном коде добавить подписки:

```
todos.OnTodoAdded += FileManager.SaveTodoList;
todos.OnTodoDeleted += FileManager.SaveTodoList;
todos.OnTodoUpdated += FileManager.SaveTodoList;
todos.OnStatusChanged += FileManager.SaveTodoList;
```

> Если в будущем появится `DatabaseManager` или `ApiManager`, можно просто подписать его вместо `FileManager` — без изменения кода классов `TodoList` и `Profile`.

4. **Проверить работу событий:**
   - При добавлении, удалении или обновлении задачи должно автоматически происходить сохранение данных в файл (через подписку).