Оглавление

[Делегаты 2](#_Toc163939780)

[Добавление методов в делегат 2](#_Toc163939781)

[Объединение делегатов 2](#_Toc163939782)

[Вызов делегата с помощью метода Invoke() 3](#_Toc163939783)

[Обобщенные делегаты (Generic) 3](#_Toc163939784)

[Делегаты как параметры методов 3](#_Toc163939785)

[Возвращение делегатов из метода 3](#_Toc163939786)

[Анонимные методы 3](#_Toc163939787)

[Лямбды 4](#_Toc163939788)

[Параметры лямбды 5](#_Toc163939789)

[Возвращение результата 5](#_Toc163939790)

[Добавление и удаление действий в лямбда-выражении, метод Invoke 5](#_Toc163939791)

[Лямбда-выражение как аргумент метода 5](#_Toc163939792)

[Лямбда-выражение как результ метода 6](#_Toc163939793)

[События 6](#_Toc163939794)

[Определение и вызов событий 6](#_Toc163939795)

[Добавление обработчика события 6](#_Toc163939796)

[Управление обработчиками (add, remove) 7](#_Toc163939797)

[Ковариантность и контравариантность делегатов 7](#_Toc163939798)

[Ковариантность 7](#_Toc163939799)

[Контрвариантность 8](#_Toc163939800)

[Ковариантность и контравариантность в обобщенных делегатах 8](#_Toc163939801)

[Ковариантность 8](#_Toc163939802)

[Контравариантность 8](#_Toc163939803)

[Делегаты Action, Predicate и Func 8](#_Toc163939804)

[Action 8](#_Toc163939805)

[Predicate 8](#_Toc163939806)

[Func 9](#_Toc163939807)

[Замыкания 9](#_Toc163939808)

[Реализация с помощью лямбда-выражений 9](#_Toc163939809)

[Применение параметров 9](#_Toc163939810)

# Делегаты

Operation operation **=** Add**;** // делегат указывает на метод Add

int result **=** operation**(**4**,** 5**);** // фактически Add(4, 5)

// можно создать объект делегата с помощью конструктора

operation **=** **new** Operation**(**Multiply**);**

result **=** operation**(**4**,** 5**);** // фактически Multiply(4, 5)

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

int Multiply**(**int x**,** int y**)** **=>** x **\*** y**;**

**delegate** int Operation**(**int x**,** int y**);**

**Делегат можно определять внутри или вне класса.**

**Методы должны соответствовать делегату по возвращаемому типу и наборы параметров (учитывая ref, out, in)**

## Добавление методов в делегат

**В делегат можно добавлять методы (с помощью +=), которые попадают в список - invokation list, и будут вызываться последовательно, но, ЕСЛИ ДЕЛЕГАТ ВОЗВРАЩАЕТ ЗНАЧЕНИЕ, то ВОЗВРАЩАЕТСЯ ЗНАЧЕНИЕ ТОЛЬКО ПОСЛЕДНЕГО МЕТОДА ИЗ СПИСКА. Можно несколько раз добавлять один и тот же метод. Можно удалять метод (если список содержит несколько одинаковых методов, то с конца списка) с помощью -=.**

Message message **=** Hello**;**

message **+=** HowAreYou**;** // теперь message указывает на два метода

message **+=** Hello**;**

message **+=** HowAreYou**;**

message **-=** Hello**;** // удалили последний Hello()

message**();** // методы вызываются по очереди

void Hello**()** **=>** Console**.**WriteLine**(**"Hello/n"**);**

void HowAreYou**()** **=>** Console**.**WriteLine**(**"How are you?/n"**);**

**delegate** void Message**();**

/\*

Hello

How are you?

How are you?

\*/

## Объединение делегатов

Message mes1 **=** Hello**;**

Message mes2 **=** HowAreYou**;**

Message mes3 **=** mes1 **+** mes2**;** // объединяем делегаты

mes3**();** // вызываются все методы из mes1 и mes2

void Hello**()** **=>** Console**.**WriteLine**(**"Hello/n"**);**

void HowAreYou**()** **=>** Console**.**WriteLine**(**"How are you?/n"**);**

**delegate** void Message**();**

/\*

Hello

How are you?

\*/

## Вызов делегата с помощью метода Invoke()

Operation**?** op **=** Add**;**

// другой способ вызова делегата - с помощью Invoke()

int result **=** op**?.**Invoke**(**4**,** 5**)** **?:** 0**;** // проверяем на null (вдруг список делегата пуст)

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

**delegate** int Operation**(**int x**,** int y**);**

## Обобщенные делегаты (Generic)

Operation**<**double**,** int**>** op **=** Square**;**

int result **=** op**(**4**);** // 16.0

double Square**(**int n**)** **=>** **(**double**)** n **\*** n**;**

**delegate** T Operation**<**T**,** K**>(**K val**);**

## Делегаты как параметры методов

DoOperation**(**5**,** 4**,** Add**);** // 9

DoOperation**(**5**,** 4**,** Multiply**);** // 20

void DoOperation**(**int a**,** int b**,** Operation op**)**

**{**

Console**.**WriteLine**(**op**(**a**,**b**));**

**}**

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

int Multiply**(**int x**,** int y**)** **=>** x **\*** y**;**

**delegate** int Operation**(**int x**,** int y**);**

**При вызове метода DoOperation мы можем передать в него в качестве третьего параметра метод, который соответствует делегату Operation.**

## Возвращение делегатов из метода

Letter op **=** SelectOp**(true);**

string result **=** op**();** // A

Letter SelectOp**(**bool a**)** **=>** a **==** "A" **?** A**()** **:** B**();**

string A**()** **=>** "A"

string B**()** **=>** "B"

**delegate** string Letter**();**

# Анонимные методы

***Анонимные методы используются для создания экземпляров делегатов.***

Operation operation **=** **delegate** **(**int x**,** int y**)**

**{**

**return** x **+** y**;**

**};**

int result **=** operation**(**4**,** 5**);** // 9

**delegate** int Operation**(**int x**,** int y**);**

***Если параметров нет, то скобки опускаются***

Message message **=** **delegate**

**{**

Console**.**WriteLine**(**"Message"**);**

**};**

message**();** // Message

**delegate** void Operation**();**

***Даже если делегат содержит параметры, то в анонимном методе они могут не использоваться, и скобки опускаются***

Operation operation **=** **delegate**

**{**

**return** 0**;**

**};**

int result **=** operation**(**4**,** 5**);** // 9

**delegate** int Operation**(**int x**,** int y**);**

***Можно передавать анонимный метод в качестве параметра метода типа делегата***

ShowMessage**(**"Hello!"**,** **delegate** **(**string mes**)**

**{**

Console**.**WriteLine**(**mes**);**

**});**

static void ShowMessage**(**string message**,** MessageHandler handler**)**

**{**

handler**(**message**);**

**}**

**delegate** void MessageHandler**(**string message**);**

# Лямбды

***(список\_параметров) => выражение***

**Лямбда-выражения представляют упрощенную запись анонимных методов. С точки зрения типа данных лямбда-выражение представляет делегат.**

Message hello **=** **()** **=>** Console**.**WriteLine**(**"Hello"**);**

hello**();** // Hello

**delegate** void Message**();**

**Если лямбда-выражение содержит несколько действий, то они помещаются в фигурные скобки:**

Message hello **=** **()** **=>**

**{**

Console**.**Write**(**"Hello "**);**

Console**.**WriteLine**(**"World"**);**

**};**

hello**();** // Hello World

**delegate** void Message**();**

**Начиная с версии C# 10:**

var hello **=** **()** **=>** Console**.**WriteLine**(**"Hello"**);**

// переменной hello присваивается тип встроенного делегата Action,

// который не принимает параметров и ничего не возвращает

hello**();** // Hello

## Параметры лямбды

**При определении списка параметров мы можем не указывать для них тип данных:**

Operation sum **=** **(**x**,** y**)** **=>** Console**.**WriteLine**(**$"{x + y}"**);**

sum**(**1**,** 2**);** // 3

**delegate** void Operation**(**int x**,** int y**);**

**Но при использовании неявной типизации (var) их нужно указать:**

var sum **=** **(**int x**,** int y**)** **=>** Console**.**WriteLine**(**$"{x + y}"**);**

**Если лямбда имеет один параметр, для которого не требуется указывать тип данных, то скобки можно опустить:**

Print print **=** message **=>** Console**.**WriteLine**(**message**);**

print**(**"Hello"**);** // Hello

**delegate** void Print**(**string message**);**

**Начиная с C# 12 параметры лямбда-выражений могут иметь значения по умолчанию:**

var welcome **=** **(**string message **=** "Hello"**)** **=>** Console**.**WriteLine**(**message**);**

welcome**();** // Hello

welcome**(**"Bye"**);** // Bye

## Возвращение результата

var sum **=** **(**int x**,** int y**)** **=>** x **+** y**;**

int result **=** sum**(**3**,** 2**);** // 5

// Если лямбда-выражение содержит несколько выражений (или одно выражение в фигурных

// скобках), тогда нужно использовать оператор return, как в обычных методах:

Operation multiply **=** **(**x**,** y**)** **=>**

**{**

**...**

**return** x **\*** y**;**

**}**

result **=** **(**3**,** 2**);** // 6

**delegate** int Operation**(**int x**,** int y**);**

## Добавление и удаление действий в лямбда-выражении, метод Invoke

var hello **=** **()** **=>** Console**.**WriteLine**(**"Hello "**);**

var world **=** **()** **=>** Console**.**WriteLine**(**"World"**);**

hello **+=** world**;**

hello **+=** **()** **=>** Console**.**WriteLine**(**"!"**);**

hello**?.**Invoke**();** // Hello World!

hello **-=** world**;**

hello**();** // Hello !

## Лямбда-выражение как аргумент метода

ShowMessage**(**"Hello!"**,** mes **=>** Console**.**WriteLine**(**mes**));**

static void ShowMessage**(**string message**,** MessageHandler handler**)**

**{**

handler**(**message**);**

**}**

**delegate** void MessageHandler**(**string message**);**

## Лямбда-выражение как результ метода

Letter op **=** SelectOp**(true);**

string result **=** op**();** // A

Letter SelectOp**(**bool a**)** **=>**

**{**

**return** a **==** "A" **?** **()** **=>** "A" **:** **()** **=>** "B"**;**

**}**

**delegate** string Letter**();**

# События

## Определение и вызов событий

**В одном классе определяем событие, и в нужных местах (в нужных методах) вызываем это событие (с разными параметрами, если нужно)**

class MyClass

**{**

**delegate** void MyDelegate**(**string message**);** // 1. Определение делегата

**event** MyDelegate MyEvent**;** // 2.Определение события

**...**

MyEvent**(**"Произошло хорошее событие"**);** // 3.Вызов события

**...**

MyEvent**?.**Invoke**(**"Произошло плохое событие"**);** // Вызов события в другом месте

**}**

## Добавление обработчика события

**В другом классе создаем обработчик события - метод с той же сигнатурой, что и делегат события, который реализует поведение-реакцию на событие (в зависимости от пришедших параметров, если они есть). Добавляем обработчик события с помощью +=**

MyClass**.**MyEvent **+=** EventHandler**;** // 5. Добавление обработчика события

MyClass**.**MyEvent **+=** EventHandlerRed**;** // Добавление другого обработчика события

MyClass**.**MyEvent **-=** EventHandler**;** // Удаление обработчика события

void EventHandler**(**string message**)** **=>** Console**.**WriteLine**(**message**);** // 4. Реализация обработчика события

void EventHandlerRed**(**string message**)** // Реализация другого обработчика события

**{**

// Устанавливаем красный цвет символов

Console**.**ForegroundColor **=** ConsoleColor**.**Red**;**

Console**.**WriteLine**(**message**);**

// Сбрасываем настройки цвета

Console**.**ResetColor**();**

**}**

***Установка в качестве обработчика делегата:***

MyClass**.**MyEvent **+=** **new** MyClass**.**MyDelegate**(**EventHandler**);** // установка делегата через конструктор

***Установка в качестве обработчика анонимного метода:***

MyClass**.**MyEvent **+=** **delegate** **(**string message**)** **=>** Console**.**WriteLine**(**message**);**

***Установка в качестве обработчика лямбда-выражения:***

MyClass**.**MyEvent **+=** message **=>** Console**.**WriteLine**(**message**);**

## Управление обработчиками (add, remove)

class MyClass

**{**

**delegate** void MyDelegate**(**string message**);** // делегат

MyDelegate**?** myEvent**;** // переменная делегата

**event** MyDelegate MyEvent // переменная события

**{**

**add** // будет вызываться при добавлении обработчика

**{**

myEvent **+=** **value;** // value - ключевое слово - добавляемый обработчик

**}**

**remove** // будет вызываться при удалении обработчика

**{**

myEvent **-=** **value;**

**}**

**}**

**...**

**}**

MyClass**.**MyEvent **+=** EventHandler**;**

void EventHandler**(**string message**)** **=>** Console**.**WriteLine**(**message**);**

# Ковариантность и контравариантность делегатов

class Message **{**

string text**;**

Message**(**string text**)** **=>** **this.**text **=** text**;**

**}**

class Email **:** Message **{** Email**(**string text**)** **:** **base(**text**)** **{}** **}**

**Дочерний класс - это частный случай родительского класса, и он может иметь больший функционал, чем родительский класс.**

**Ковариантность - это возвращение дочерних классов вместо родительских.**

**(могу вернуть Email вместо Message, т.к. Email - это и есть Message, но не наоборот, и то, что есть в Message, есть и в Email).**

**Контрвариантность - это использование родительских классов вместо дочерних.**

**(в качестве реализации делегата могу использовать метод, принимающий Message вместо Email, т.к. то, что есть в Message, есть и в Email).**

## Ковариантность

**Ковариантность позволяет передать делегату метод, возвращаемый тип которого является дочерним от возвращаемого типа делегата.**

**delegate** Message MyDelegate**(**string text**);** // делегат возвращает родителя

Email MyMethod**(**string text**)** **=>** **new** Email**(**text**);** // метод возвращает экземпляр дочернего //класса

MyDelegate myDelegate **=** MyMethod**;** // ковариантность

## Контрвариантность

**delegate** void MyDelegate**(**Email message**);** // делегат принимает экземпляр дочернего класса

void MyMethod**(**Message message**);** // метод принимает родителя

MyDelegate myDelegate **=** MyMethod**;** // контравариантность

## Ковариантность и контравариантность в обобщенных делегатах

out - можно этот тип или родительский

in - можно этот тип или дочерний

### Ковариантность

**delegate** T MyDelegate**<out** T**>(**string text**);** // делегат

// реализация делегата возвращает экземпляр дочернего типа

MyDelegate**<**Email**>** myMethod1 **=** text **=>** **new** Email**(**text**);**

// реализация делегата возвращает экземпляр родительского типа

MyDelegate**<**Message**>** myMethod2 **=** myDelegate1**;** // ковариантность

Message message **=** myMethod2**(**"Message"**);** // вызов делегата. Возвращение родительского //типа

### Контравариантность

**delegate** void MyDelegate**<in** T**>(**T message**);**

// реализация делегата принимает кземпляр родительского типа

MyDelegate**<**Message**>** myMethod1 **=** message **=>** Console**.**WriteLine**(**message**.**text**);**

// реализация делегата принимает кземпляр дочернего типа

MyDelegate**<**Email**>** myMethod2 **=** myDelegate1**;** // контравариантность

myMethod2**(new** Email**(**"Email"**));** // вызов делегата. Использование дочернего типа

# Делегаты Action, Predicate и Func

## Action

**public** **delegate** void Action**()**

**public** **delegate** void Action**<in** T**>(**T obj**)**

**Принимает от 0 до 16 объектов, ничего не возвращает.**

void Add**(**int x**,** int y**)** **=>** Console**.**WriteLine**(**$**{**x **+** y**});** // метод соответсвует делегату Action

void DoOperation**(**int a**,** int b**,** Action**<**int**,** int**>** operation**)** // метод принимает делегат Action

**=>** operation**(**a**,** b**);**

DoOperation**(**5**,** 6**,** Add**);** // 11

## Predicate

**delegate** bool Predicate**<in** T**>(**T obj**);**

**Принимает один параметр и возвращает значение типа bool.**

Predicate**<**int**>** isPositive **=** **(**int x**)** **=>** x **>** 0**;** // проверяет на положительность

## Func

TResult Func**<out** TResult**>()**

TResult Func**<in** T**,** **out** TResult**>(**T obj**)**

**Принимает от 0 до 16 объектов, возвращает результат.**

int Add**(**int x**,** int y**)** **=>** x **+** y**;** // метод соответсвует делегату Func

int DoOperation**(**int a**,** int b**,** Func**<**int**,** int**>** operation**)** // метод принимает делегат Func

**=>** operation**(**a**,** b**);**

int result **=** DoOperation**(**5**,** 6**,** Add**);** // 11

# Замыкания

**Замыкание (closure) представляет объект функции, который запоминает свое лексическое окружение и может его менять даже при вызове вне своей области видимости.**

**Технически замыкание включает три компонента:**

**- внешняя функция, которая определяет некоторую область видимости**

**- переменные и параметры (лексическое окружение), которые определены во внешней функции**

**- вложенная функция, которая использует переменные и параметры внешней функции**

Action Outer**()** // метод или внешняя функция

**{**

int x **=** 5**;** // лексическое окружение - локальная переменная

void Inner**()** // локальная функция

**{**

x**++;** // операции с лексическим окружением

Console**.**WriteLine**(**x**);**

**}**

**return** Inner**;** // возвращаем локальную функцию

**}**

Action fn **=** Outer**();** // fn = Inner, так как метод Outer возвращает функцию Inner

// вызываем внутреннюю функцию Inner

fn**();** // 6

fn**();** // 7

fn**();** // 8

## Реализация с помощью лямбда-выражений

Action outerFun **=** **()** **=>**

**{**

int x **=** 5**;**

var innerFun **=** **()** **=>** Console**.**WriteLine**(++**x**);**

**return** innerFun**;**

**};**

Action fn **=** outerFun**();** // fn = innerFun, так как outerFun возвращает innerFn

// вызываем innerFun

fn**();** // 6

fn**();** // 7

fn**();** // 8

## Применение параметров

Func**<**int**,** int**>** Multiply**(**int n**)**

**{**

int Inner**(**int m**)** **=>** n **\*** m**;**

**return** Inner**;**

**}**

Func**<**int**,** int**>** func **=** Multiply**(**5**);**

// var multiply = (int n) => (int m) => n \* m;

// var func = multiply(5);

Console**.**WriteLine**(**func**(**5**));** // 25

Console**.**WriteLine**(**func**(**6**));** // 30

Console**.**WriteLine**(**func**(**7**));** // 35