

Министерство труда и социальной защиты  
Республики Беларусь

Государственное учреждение образования  
«Республиканский институт повышения квалификации  
и переподготовки работников Министерства труда  
и социальной защиты Республики Беларусь»

П. С. Карпович

# С# в примерах

## Практикум

В двух частях

Часть 2

Минск  
Минский государственный ПТК полиграфии  
2019

УДК 004(075,9)  
ББК 32,973,26-018,2я75  
К26

*Рекомендовано к изданию Советом института государственного учреждения образования «Республиканский институт повышения квалификации переподготовки работников Министерства труда и социальной защиты Республики Беларусь», протокол от 31.10.2019 № 4.*

Автор: *П. С. Карпович*, преподаватель кафедры информационных технологий РИПК Минтруда и соцзащиты.

Рецензенты: старший преподаватель кафедры защиты информации учреждения образования «Белорусский государственный университет информатики и радиоэлектроники» *Г. А. Пухир*, программист ООО «Легат Бай» *А. О. Шашков*.

**Карпович, П. С.**

К26    С# в примерах : практикум. В 2 ч. Ч. 2 / П. С. Карпович. – Минск : Минский государственный ПТК полиграфии, 2019. – 108 с.  
ISBN 978-985-7249-06-0.

Практикум «С# в примерах» рекомендовано использовать для слушателей системы повышения квалификации по информационным технологиям, а также при проведении занятий по дисциплинам переподготовки по специальности «Программное обеспечение информационных систем»

УДК 004(075,9)  
ББК 32,973,26-018,2я75

**ISBN 978-985-7249-06-0 (ч. 2)**  
**ISBN 978-985-7249-07-7**

© Карпович П. С., 2019  
© РИПК Минтруда и соцзащиты, 2019  
© Оформление. УО «Минский государственный ПТК полиграфии», 2019

## ОГЛАВЛЕНИЕ

Оглавление.....	3
ВВЕДЕНИЕ.....	4
Глава 1. ОБОБЩЕНИЯ.....	5
Глава 2. ОПЕРАТОРЫ.....	31
Глава 3. ДЕЛЕГАТЫ, ЛЯМБДЫ И СОБЫТИЯ .....	57
Приложение А .....	94
Приложение Б .....	97
Приложение В .....	102
ИСПОЛЬЗУЕМАЯ ЛИТЕРАТУРА .....	107

## ВВЕДЕНИЕ

Практикум «С# в примерах» предназначен для слушателей системы повышения квалификации по информационным технологиям, а также при проведении занятий по дисциплинам переподготовки по специальности «Программное обеспечение информационных систем».

Основными задачами практикума являются:

- ознакомление слушателей с синтаксисом и семантикой языка программирования С#;
- описание особенностей архитектуры .NET;
- формирование навыков разработки приложений в рамках парадигмы объектно-ориентированного программирования.

В языке С#, созданном компанией Microsoft для поддержки среды .NET Framework, проверенные временем средства программирования усовершенствованы с помощью самых современных технологий. С# предоставляет очень удобный и эффективный способ написания программ для современной среды вычислительной обработки данных, которая включает операционную систему Windows, Интернет, компоненты и пр.

С# – это язык, разработанный Эндерсом Хейлсбергом в корпорации Microsoft в качестве основной среды разработки для .Net Framework и всех будущих продуктов Microsoft. С# основан на других языках, в основном, С++, Java, Delphi, Modula-2 и Smalltalk. Характеризуя основные особенности языка, отметим, что с одной стороны, для С# в еще большей степени, чем для упомянутых выше языков, характерна внутренняя объектная ориентация; с другой стороны, в нем реализована концепция упрощения объектов, что существенно облегчает освоение мира объектно-ориентированного программирования.

Практикум состоит из двух частей и ряда приложений.

Первая часть практикума содержит шесть глав и пять приложений.

Первая глава первой части представляет собой введение, где разбираются особенности архитектуры .NET, кратко описываются ее основные компоненты.

Вторая глава содержит основы языка С#.

Третья и четвертая главы посвящены принципам объектно-ориентированного программирования, приводится понятие классов и объектов.

В пятой и шестой главе рассматриваются реализация наследования и полиморфизма в языке программирования.

В приложениях представлены примеры реализации программ.

Вторая часть практикума состоит из трех глав и трех приложений.

В первой главе второй части рассматривается создание обобщенных типов и методов.

Вторая глава этой части практикума посвящена составу операторов языка С#, их синтаксису и семантике.

В третьей главе второй части представлено понятие делегатов и их использование.

В приложениях также приведены примеры реализации программ.

## Глава 1. ОБОБЩЕНИЯ

### Задача

Представим, что нам требуется создать класс коллекции, например, собственную реализацию циклического списка.

```
// Класс Циклического списка
public class CyclicalList
{
    // Вложенный приватный класс Элемент
    private class Node
    {
        // Ссылка на следующий элемент
        public Node Next { get; set; }
        // Данные элемента типа int
        public int Data { get; set; }
    }

    // Ссылка на текущий указатель списка
    private Node head;

    // Свойство для количества элементов в списке
    public int Count { get; set; }

    // Конструктор по умолчанию
    public CyclicalList()
    {
        // Задаёт ссылку нулевой
        head = null;
        // И количество элементов - 0
        Count = 0;
    }

    // Метод для добавления нового числа в список
    public void Add(int newVal)
    {
        // Создаём новый объект класса Элемент списка
        Node n = new Node();
        // Записываем в него числовое значение
        n.Data = newVal;
        // Если в списке ещё нет ни одного элемента
        if (head == null)
        {
            // Зацикливаем новый элемент
            n.Next = n;
            // Устанавливаем указатель списка на него
            head = n;
        }
        else
        {
            // Ссылка на следующий элемент
            head.Next = n;
            // Увеличиваем количество элементов
            Count++;
        }
    }
}
```

```

// Иначе ставим следующим после нового элемента элемент, следующий за
указателем списка
    n.Next = head.Next;
// А следующим за указателем списка - новый элемент
    head.Next = n;
}
// Увеличиваем количество элементов в списке
    ++Count;
}

// Метод для перемещения указателя списка и получения значения
public int Next()
{
    // Устанавливаем указатель списка на следующий элемент за текущим
    указателем списка
        head = head.Next;
// Возвращаем значение из него
    return head.Data;
}
}

class Program
{
    static void Main(string[] args)
    {
        // Создаём объект циклического списка
        CyclicalList cList = new CyclicalList();
        // Добавляем в него разные числа
        cList.Add(1);
        cList.Add(2);
        cList.Add(3);
        cList.Add(4);
        cList.Add(5);
        cList.Add(6);
        // По циклу, для количества элементов в списке
        for (int i = 0; i < cList.Count; ++i)
        {
            // Выводим каждый последующий элемент на консоль
            Console.Write("{0} ", cList.Next());
        }
    }
}

```

## Проблема

И что же в результате?

Класс работает, все отлично, но... Он работает только для целых чисел типа int.

А что, если требуется сделать список строк? Можно создать такой же класс, только для строк...

А потом такой же для типа double, дат, вашего собственного класса и т.д.

**Проблему нужно решать**, для чего нам нужны обобщенные типы!

## Обобщения

Обобщения (или шаблоны) – это параметризованные типы.

Механизм параметризованных типов позволяет создавать классы, структуры, интерфейсы и методы, в которых используемые типы данных могут указываться в виде параметров.

## Форма объявления параметризованного класса:

```
class Имя_класса <список_параметров_типов>
{
    // Здесь можно использовать типы из списка параметров
}
```

## Пример

Ниже создается параметризованный (обобщенный) класс с типом-параметров T:

```
class GenericClass <T>
{
    // ...
}
```

А здесь обобщенный интерфейс:

```
interface IGeneric<T>
{
    // ...
}
```

Имя первого параметра-типа принято называть заглавной буквой T.

## Обобщенные типы

Когда у имени класса после имени стоят треугольные скобки с параметром (T или другим), такой класс называется обобщенным классом. По аналогии с классом бывают обобщенные структуры, обобщенные интерфейсы и обобщенные методы.

В случае с обобщенным методом тип-параметр также указываем в треугольных скобках после имени метода:

```
public int GenericMethod<T>(float param1, bool param2)
{
    // ...
}
```

## Использование типа-параметра

Когда тип описан параметризированным, внутри него можно использовать данный тип Т (или другое имя), как будто это обычный определенный тип.

```
// Обобщённый класс MyGeneric
class MyGeneric<T>
{
    // Свойство целочисленного типа
    public int IntProperty { get; set; }

    // Поле типа-параметра
    private T field;
    // Свойство типа-параметра для этого поля
    public T Property
    {
        get { return field; }
        set { field = value; }
    }

    // Конструктор, принимающий число и объект типа-параметра
    public MyGeneric(int a, T b)
    {
        IntProperty = a;
        Property = b;
    }

    // Метод, возвращающий объект типа-параметра
    public T PowerfulMethod(char a)
    {
        // ...
        return Property;
    }
}
```

## Использование обобщений

```
List<int> ls = new List<int>();

Dictionary<string, string> dict = new Dictionary<string, string>();

Stack<double> st1 = new Stack<double>();
```

При создании объекта класса (структуры) после имени класса в треугольных скобках записывается конкретное значение типа для типа-параметра. И объект создается именно для работы с указанным типом.

## Обобщения в CLR

Фактически использование обобщенного класса в программном коде с указанием какого-либо конкретного типа приводит к созданию отдельного класса, специально для этого типа.



```

// Обобщённый класс MyObj с типом-параметром T
class MyObj<T>
{
    // Приватное поле типа типа-параметра
    private T obj;

    // Конструктор
    public MyObj(T obj)
    {
        this.obj = obj;
    }

    // Метод для вывода на консоль информации о типе-параметре
    public void ShowParamType()
    {
        Console.WriteLine("Type parameter: " + typeof(T));
    }

    // Метод для вывода на консоль информации о типе класса
    public void ShowClassType()
    {
        Console.WriteLine("Generic class type: " + this.GetType() +
"\n");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Создаём класс MyObj для типа int
        MyObj<int> intGeneric = new MyObj<int>(2);
        // Смотрим на его типы
        intGeneric.ShowParamType();
        intGeneric.ShowClassType();

        // Создаём класс MyObj для типа string
        MyObj<string> stringGeneric = new MyObj<string>("dwada");
        // Смотрим на его типы
        stringGeneric.ShowParamType();
        stringGeneric.ShowClassType();

        // Создаём класс MyObj для типа Program
        MyObj<Program> programGeneric = new MyObj<Program>(new
Program());
        // Смотрим на его типы
        programGeneric.ShowParamType();
        programGeneric.ShowClassType();
    }
}

```

## Обобщения в CLR

В этом примере среда CLR сама создаст в памяти три варианта класса для используемых типов, код обобщения будет выступать для нее в качестве шаблона.

Фактически, получатся вот такие классы:

```
// Класс MyObj для int-a
class MyObj<int>
{
    // Поле типа int
    private int obj;

    // Конструктор с параметром int
    public MyObj(int obj)
    {
        this.obj = obj;
    }

    public void ShowParamType()
    {
        Console.WriteLine("Type parameter: " + typeof(int));
    }

    public void ShowClassType()
    {
        Console.WriteLine("Generic class type: " + this.GetType() +
"\n");
    }
}

// Класс MyObj для string-a
class MyObj<string>
{
    // Поле типа string
    private string obj;

    // Конструктор с параметром string
    public MyObj(string obj)
    {
        this.obj = obj;
    }

    public void ShowParamType()
    {
        Console.WriteLine("Type parameter: " + typeof(string));
    }

    public void ShowClassType()
    {
        Console.WriteLine("Generic class type: " + this.GetType() +
"\n");
    }
}
```

```
// Класс MyObj для класса Program
class MyObj<Program>
{
    // Поле типа Program
    private Program obj;

    // Конструктор с параметром типа Program
    public MyObj(Program obj)
    {
        this.obj = obj;
    }

    public void ShowParamType()
    {
        Console.WriteLine("Type parameter: " + typeof(Program));
    }

    public void ShowClassType()
    {
        Console.WriteLine("Generic class type: " + this.GetType() +
"\n");
    }
}
```

### Обобщенный циклический список

Теперь можно исправить первый пример, и написать полноценный класс универсального циклического списка.

```
// Обобщённый класс Циклического списка с типом-параметром T
public class CyclicalList<T>
{
    private class Node
    {
        public Node Next { get; set; }
        // Теперь данные в элементе представлены в виде типа T
        public T Data { get; set; }
    }

    private Node head;

    public int Count { get; set; }

    public CyclicalList()
    {
        head = null;
        Count = 0;
    }

    // Метод добавления элемента принимает значение типа типа-параметра T
    public void Add(T newVal)
    {

```

```

        Node n = new Node();
// Записывает его в Элемент списка
n.Data = newVal;
if (head == null)
{
    n.Next = n;
    head = n;
}
else
{
    n.Next = head.Next;
head.Next = n;
}
++Count;
}

// Метод теперь возвращает объект типа T
public T Next()
{
    head = head.Next;
return head.Data;
}
}

// Просто некий класс
class Auto
{
// ...
}

class Program
{
static void Main(string[] args)
{
// Создаём объект циклического списка для типа int
CyclicalList<int> iList = new CyclicalList<int>();
// Добавляем в него разные числа
iList.Add(1);
iList.Add(2);
iList.Add(3);
iList.Add(4);
iList.Add(5);
iList.Add(6);
for (int i = 0; i < iList.Count; ++i)
{
// Выводим каждый последующий элемент на консоль
Console.Write("{0} ", iList.Next());
}
Console.WriteLine();

// Создаём объект циклического списка для типа string

```

```

Cyclicallist<string> sList = new Cyclicallist<string>();
// Этот список должен содержать строки
    sList.Add("white");
sList.Add("brown");
    sList.Add("red");
    sList.Add("yellow");
for (int i = 0; i < sList.Count; ++i)
{
    // Выводим каждый последующий элемент на консоль
    Console.Write("{0} ", sList.Next());
}
    Console.WriteLine();

// Создаём объект циклического списка для типа Auto
Cyclicallist<Auto> aList = new Cyclicallist<Auto>();
// Добавляем в него объекты класса Auto
aList.Add(new Auto());
    aList.Add(new Auto());
    aList.Add(new Auto());
for (int i = 0; i < aList.Count; ++i)
{
    // Выводим каждый последующий элемент на консоль
    Console.Write("{0} ", aList.Next());
}
    Console.WriteLine();
}
}

```

### Статические члены обобщенных классов

Статические члены обобщенных классов требуют особого внимания. Из-за того, что для обобщений с конкретными типами создаются отдельные классы, для каждого из таких классов будут существовать свои отдельные статические члены.

```

// Обобщённый класс
publicclass StaticDemo<T>
{
    // Статическое свойство
    publicstaticint X { get; set; }
}

staticvoid Main(string[] args)
{
    // Одно статическое свойство
    StaticDemo<string>.X = 4;
    // Другое статическое свойство
    StaticDemo<int>.X = 5;
    // Выведет на консоль 4
    Console.WriteLine(StaticDemo<string>.X);
}

```

## Начальные значения неопределённых типов

В конструкторах принято инициализировать свойства класса/структуры.

А как быть если нужно инициализировать свойство типа-параметра?

В обобщении мы не можем знать, каким типом он будет выступать.

Типам значений можно присвоить 0, но нельзя присвоить null, а ссылочным типам, наоборот: можно присвоить null, но нельзя присвоить 0.

## Ключевое слово default

Для этих целей существует ключевое слово default, которое позволяет назначить переменной значение по умолчанию конкретно для ее типа.

```
// Обобщённый класс ObEx
class ObEx<T>
{
    // Свойство типа T
    public T Obj { get; set; }

    // Конструктор по умолчанию
    public ObEx()
    {
        // Свойство неопределённого типа инициализируется значением по умолчанию
        Obj = default(T);
    }
}

class Program
{
    static void Main()
    {
        // Создаём объект класса ObEx для ссылочного типа Random
        ObEx<Random> a = new ObEx<Random>();
        // Создаём объект класса ObEx для типа по значению int
        ObEx<int> b = new ObEx<int>();

        // Если значение null, выводим сообщение об этом
        if (a.Obj == null)
        {
            Console.WriteLine("a.obj = null");
        }
        // Если значение 0, выводим сообщение об этом
        if (b.Obj == 0)
        {
            Console.WriteLine("b.obj = 0");
        }

        Console.ReadKey(true);
    }
}

// Обобщённый класс ObEx
class ObEx<T>
{
```

```

// Свойство типа T
public T Obj { get; set; }

// Конструктор по умолчанию
public ObEx()
{
    // Свойство неопределённого типа инициализируется значением по умолчанию
    Obj = default(T);
}

class Program
{
    static void Main()
    {
        // Создаём объект класса ObEx для ссылочного типа Random
        ObEx<Random> a = new ObEx<Random>();
        // Создаём объект класса ObEx для типа по значению int
        ObEx<int> b = new ObEx<int>();

        // Если значение null, выводим сообщение об этом
        if (a.Obj == null)
        {
            Console.WriteLine("a.obj = null");
        }
        // Если значение 0, выводим сообщение об этом
        if (b.Obj == 0)
        {
            Console.WriteLine("b.obj = 0");
        }

        Console.ReadKey(true);
    }
}

```

## Неопределенность типа-параметра

Поскольку до момента выполнения программы не известно, какой конкретно тип будет использоваться в качестве типа-параметра, накладываются большие ограничения по его использованию.

Чем шире возможный диапазон допустимых типов, тем меньше между ними такого общего, что может использоваться с данным типом. Так, по умолчанию, для типа T доступны только основные методы класса Object и даже нельзя создать его экземпляр.

## Ограничения типа-параметра

Но на тип-параметр можно накладывать определенные ограничения, тем самым, во-первых, сужая для него перечень возможных типов, и, во-вторых, уточняя этот тип, т.е. позволяя использовать с ним больший набор операций.

## Виды ограничений

Таблица 7.1

### Ограничения типа параметра

Синтаксис ограничения	Описание
whereT : struct	Допускаются только структуры
whereT : class	Допускаются только классы
whereT : имя_базового_класса	Допускаются только потомки указанного класса
whereT : имя_интерфейса	Допускаются только классы, реализующие указанный интерфейс
whereT : new()	Допускаются только типы с конструктором по умолчанию
whereT : unmanaged	Допускаются только структуры без ссылок внутри

### Демонстрация ограничений

```
// Класс Работник
publicclass Employee
{
    // Свойство Идентификатор
    publicint ID { get; set; }
    // Свойство Имя
    publicstring Name { get; set; }

    // Конструктор
    public Employee(string s, int i)
    {
        Name = s;
        ID = i;
    }
}

// Обобщённый класс Списка Работников
// Наложённое ограничение указывает, что в качестве типа T
// может выступать только производный класс от класса Работник или он
сам
publicclass EmployeesList<T>where T : Employee
{
    // Приватный класс Элемента Списка
    privateclass Node
    {
        // Конструктор
        public Node(T t)
        {
            Next = null;
        }
    }
}
```



```

        Data = t;
    }

    // Свойства для ссылки и для данных
    public Node Next { get; set; }
    public T Data { get; set; }
}

// Приватная ссылка на начало списка
private Node head;

// Метод для добавления нового элемента в начало списка
public void AddHead(T t)
{
    Node n = new Node(t);
    n.Next = head;
    head = n;
}

// Метод для поиска в списке первого Работника с заданным именем
public T FindFirstOccurrence(string s)
{
    // Ссылка на начало списка
    Node current = head;
    // Ссылка на объект типа-параметра
    T t = null;

    // Пока не дойдём до конца списка
    while (current != null)
    {
        // Благодаря ограничению мы можем получить доступ к содержимому класса
        Employee в
        // объекте типа T, и проверить свойство Name
        if (current.Data.Name == s)
        {
            // Если имя совпало, запоминаем этого работника
            t = current.Data;
            // И выходим из цикла
            break;
        }
        else
        {
            // Или переходим к следующему работнику
            current = current.Next;
        }
    }
    // Возвращаем объект t (null или найденный работник)
    return t;
}
}

```

## Ещё один пример

```
// Интерфейс пользовательских данных
interface IUserInfo
{
    // Свойства для Имени и Возраста пользователя
    string Name { get; set; }
    int Age { get; set; }
}

// Класс Базовой информации о пользователе
// Реализует интерфейс IUserInfo
class SimpleUserInfo : IUserInfo
{
    // Свойства Имя и Возраст
    public string Name { get; set; }
    public int Age { get; set; }

    // Конструктор
    public SimpleUserInfo(string name, int age)
    {
        this.Name = name;
        this.Age = age;
    }
}

// Класс Полной информации о пользователе
// Также реализует интерфейс IUserInfo
class FullUserInfo : IUserInfo
{
    // Свойства Имя, Возраст и Семья
    public string Name { get; set; }
    public int Age { get; set; }
    public string Surname { get; set; }

    // Конструктор
    public FullUserInfo(string surname, string name, int age)
    {
        this.Name = name;
        this.Age = age;
        this.Surname = surname;
    }

    // Переопределён метод ToString
    public override string ToString()
    {
        // Формирует и возвращает строку из всех свойств этого класса
        string s = String.Format("Информация о пользователе: \n{0} {1} {2}\n",
            Name, Surname, Age);
        return s;
    }
}
```

```

}

// Обобщенный класс, использующий ограничение на реализуемый интерфейс
// Объект этого класса можно создать только для типов, которые реализуют
интерфейс IUserInfo
class Info<T> where T : IUserInfo
{
    // Два частных поля: массив из объектов T и фактическое количество
элементов в нём
    private T[] userList;
    private int i;

    // Конструктор по умолчанию
    public Info()
    {
        // Создаём массив из 3 элементов типа T
        userList = new T[3];
        // Изначально количество элементов равно 0
        i = 0;
    }

    // Метод Добавления нового элемента
    public void Add(T obj)
    {
        // Если ещё не 3 элемента
        if (i < 3)
        {
            // В i-ый элемент массива записываем переданный
            userList[i] = obj;
            // Увеличиваем количество/индекс
            i++;
        }
    }

    // Метод для вывода массива на консоль
    public void ShowAll()
    {
        // Перебираются и выводятся элементы в массиве
        foreach (T t in userList)
        {
            Console.WriteLine(t.ToString());
        }
    }
}

class Program
{
    static void Main()
    {
        // Создаём объект Info для типа FullUserInfo
        Info<FullUserInfo> database1 = new Info<FullUserInfo>();
        // Добавляем в него 3 объекта
    }
}

```

```

        database1.Add(new FullUserInfo(name: "Marlyn", surname: "Erwyn",
age: 26));
        database1.Add(new FullUserInfo(name: "Lex", surname: "Freeday",
age: 28));
        database1.Add(new FullUserInfo(name: "Amanta", surname:
"Joseph", age: 50));
        // Вызываем метод вывода на консоль
        database1.ShowAll();

        Console.ReadKey(true);
    }
}

```

## Комплексные ограничения

Любые ограничения можно комбинировать между собой. В случае с ограничением на реализуемый интерфейс, можно, например, указывать любое количество интерфейсов.

```

class EmployeeList<T> where T : Employee, IEmployee,
System.IComparable<T>, new()
{
    // ...
}

```

## Обобщенные методы

Обобщенные методы могут быть определены внутри необобщенного класса.

### Пример

```

// Лающий интерфейс
interface IBarkable
{
    // ...
}

// Класс собаки
class Dog : IBarkable
{
    // ...
}

// Класс полицейской собаки
class PoliceDog : Dog
{
    // ...
}

// Класс собаки-космонавта
class SpaceDog : Dog
{
    // ...
}

```

```

}

// Обычный класс фабрики по производству собак
class DogFabric
{
// Обобщённый метод создания собаки, возвращает объект типа-параметра T
// Ограничение: T может быть только типом собаки и должен иметь
конструктор по умолчанию
public T CreateDog<T>()
where T : Dog, new()
{
    Console.WriteLine("Creating new {0}...", typeof(T));
    Thread.Sleep(1000);
    Console.WriteLine("All right, it's done!");
// Создаём новый объект типа T и возвращаем его
return new T();
}

// Обычный метод создания собаки, возвращает объект собаки
public Dog JustCreateDog()
{
    Console.WriteLine("Creating new {0}...", typeof(Dog));
    Thread.Sleep(1000);
    Console.WriteLine("All right, it's done!");
// Создаём новый объект типа собаки и возвращаем его
return new Dog();
}

}

class Program
{
static void Main(string[] args)
{
// Создаём фабрику собак
DogFabric fabric = new DogFabric();

// Вызываем метод простого создания собак
// И принимаем новый экземпляр
Dog dog1 = fabric.JustCreateDog();
// Этот метод не может создать ничего другого, только обыкновенных собак

// Вызываем метод фабрики по созданию собаки, указывая в качестве типа-
параметра тип обычной собаки Dog
// Принимаем экземпляр обычной собаки, которую вернул метод
Dog dog2 = fabric.CreateDog<Dog>();
// А теперь указываем в качестве типа-параметра тип полицейской собаки
PoliceDog
// И получаем объект полицейской собаки
PoliceDog dog3 = fabric.CreateDog<PoliceDog>();
// Точно так же с космической собакой
SpaceDog dog4 = fabric.CreateDog<SpaceDog>();
}
}

```

## Еще один пример

```
// Обыкновенный необобщённый класс InfoObject
class InfoObject
{
    // Обобщенный метод с ограничением на производный тип
    public static string Info<T>(T obj)
    where T : User
    {
        return obj.ToString();
    }
}

// Класс Пользователя
class User
{
    // Свойства Имени и Возраста
    public string Name { get; set; }
    public int Age { get; set; }

    // Конструктор
    public User(string Name, int Age)
    {
        this.Name = Name;
        this.Age = Age;
    }
}

// Класс Пользователя с паролем, дочерний от класса Пользователь
class UserWithPass : User
{
    // Дополнительное свойство - пароль
    public string Password { get; set; }

    // Конструктор
    public UserWithPass(string Name, int Age, string Pass)
        : base(Name, Age)
    {
        Password = Pass;
    }

    // Переопределён метод ToString
    public override string ToString()
    {
        // @ перед строкой позволяет записывать многострочные строки и
        // игнорировать спец.символы
        return String.Format(@"Информация о пользователе:
        *****
        Имя: {0}
        Возраст: {1}
        Пароль: {2}", Name, Age, Password);
    }
}
```

```

class Program
{
    static void Main()
    {
        // Создаём пользователя с очень надёжным паролем
        UserWithPass user1 = new UserWithPass(Name: "Martin", Age: 26, Pass:
"12345");

        // Вызываем обобщенный метод для класса UserPass
        string s = InfoObject.Info<UserWithPass>(user1);
        Console.WriteLine(s);

        Console.ReadKey(true);
    }
}

```

## Обобщенные интерфейсы

```

// Объявляем обобщенный интерфейс
public interface IShow<T>
where T : struct
{
    void Rewrite();
}

// Реализуем интерфейс в классе MyObj с ограничением допустимых типов на
структуры
class MyObj<T> : IShow<T> where T : struct
{
    // Свойство для хранения количества элементов массива
    public int LongOb { get; set; }
    // Приватное поле для самого массива
    private T[] myarr;

    // Конструктор с одним параметром
    public MyObj(int i)
    {
        LongOb = i;
    }

    // Конструктор с двумя параметрами
    public MyObj(int i, T[] arr)
    {
        // Записываем размер
        LongOb = i;
        // Создаём новый массив указанного размера
        myarr = new T[i];
        // Копируем значения из входного массива во внутренний
        for (int j = 0; j < arr.Length; j++)
            myarr[j] = arr[j];
    }
}

```

```

// Метод из интерфейса IShow
public void Rewrite()
{
    // Выводим всё на консоль
    Console.WriteLine("Тип: {0}", typeof(T));
    Console.WriteLine("Массив объектов: ");
    foreach (T t in myarr)
        Console.WriteLine("{0}\t", t);
    Console.WriteLine("\n");
}

}

class Program
{
    static void Main()
    {
        // Создаём массив байт
        byte[] MyArrByte = new byte[5] { 4, 5, 18, 56, 8 };
        // Создаём объект класса MyObj для байт, и передаём в его конструктор
        массив и его размер
        MyObj<byte> ByteConst = new MyObj<byte>(MyArrByte.Length, MyArrByte);
        // Вызываем метод вывода на консоль
        ByteConst.Rewrite();

        // Создаём массив числе с плавающей точкой
        float[] MyArrFloat = new float[8] { 12.0f, 1f, 3.4f, 2.8f, -334.7f, -2f,
        7.89f, 0 };
        // Создаём объект класса MyObj для float, и передаём в его конструктор
        массив и его размер
        MyObj<float> FloatConst = new MyObj<float>(MyArrFloat.Length,
        MyArrFloat);
        // Вызываем метод вывода на консоль
        FloatConst.Rewrite();

    }
}

```

## Пример

Сделав интерфейс обобщенным, можно, например, описать в нем такой метод, который будет в качестве типа возвращаемого значения иметь тип метода, в котором этот интерфейс реализовывается.

```

// Обобщённый интерфейс с типом-параметром T
// С ограничением, что тип T должен быть классом, реализующим этот
интерфейс
interface ICloneable<T> where T : class, ICloneable<T>
{
    // Метод возвращает объект типа T
    public T Clone();
}

```



```
// Класс Foo, который реализует этот обобщённый интерфейс и в качестве
// типа-параметра использует свой тип Foo
class Foo : ICloneable<Foo>
{
    // Значит, этот метод должен возвращать объект этого же типа Foo
    public Foo Clone()
    {
        // ...
    }
}
```

## Ковариантность и контравариантность

Понятие ковариантности связано с возможностью методов обобщенного интерфейса использовать значения разных типов, связанных между собой наследованием.

Есть три варианта такого поведения:

- инвариантность (по умолчанию);
- ковариантность (ключевое слово `out`);
- контравариантность (ключевое слово `in`).

Упрощая, можно сказать, что ковариантность позволяет настраивать полиморфизм для обобщенных интерфейсов (и делегатов<sup>1</sup>).

## Инвариантность интерфейсов

Инвариантность обобщенных интерфейсов заключается в отсутствии возможности использовать полиморфизм для конкретных типов со связанными типами параметрами.

```
// Класс Банковского счёта
class Account
{
    // Свойство Баланс на счету
    public decimal Balance { get; set; }

    // Метод Перевода суммы на этот счёт
    public void DoTransfer(decimal sum)
    {
        Console.WriteLine("Transferring {0}$...", sum);
        Balance += sum;
    }
}

// Производный класс от Банковского счёта - депозит
class DepositAccount : Account
{
    // ...
}
```

---

<sup>1</sup> см. ниже, гл.3

```

}

// Обобщённый интерфейс Банка с типом-параметром T
interface IBank<T>
{
// Метод для создания аккаунта, возвращает объект типа T
    T CreateAccount(int sum);
}

// Обобщённый класс Банк с типом-параметром T
// Реализует интерфейс Банка с этим же типом-параметром
// И на тип-параметр наложены ограничение на наличие конструктора по
умолчанию
// и соответствие классу Account, либо производным от него классам
class Bank<T> : IBank<T>
where T : Account, new()
{
// Реализация метода CreateAccount из интерфейса IBank
public T CreateAccount(int sum)
{
// Создаём новый счёт
    T acc = new T();
// Выполняем начальный перевод
    acc.DoTransfer(sum);
// Возвращаем объект счёта
return acc;
}
}

class Program
{
static void Main(string[] args)
{
// Создадим объект Банка для типа DepositAccount
// Полученный класс будет реализовывать интерфейс IBank<DepositAccount>
IBank<DepositAccount> depositBank = new Bank<DepositAccount>();
Account acc1 = depositBank.CreateAccount(34);

// Ошибка
// Мы не можем представить этот объект в виде интерфейса IBank<Account>
// Потому что depositBank реализует интерфейс IBank<DepositAccount>
// А IBank<Account> и IBank<DepositAccount> - это два разных интерфейса
IBank<Account> ordinaryBank = depositBank;
    Account acc2 = ordinaryBank.CreateAccount(45);

}
}

```

## Ковариантность интерфейсов

Ковариантность интерфейса позволяет представлять объекты с интерфейсом с более конкретным типом в виде интерфейсов с менее конкретным типом. Ковариантные типы в методах интерфейса можно использовать только для возвращаемых значений в методах.

```
Class Account
{
    publicdecimal Balance { get; set; }

    publicvoid DoTransfer(decimal sum)
    {
        Console.WriteLine("Transferring {0}$....", sum);
        Balance += sum;
    }
}

classDepositAccount : Account
{
    // ...
}

// Интерфейс с ковариантным параметром типа
interfaceIBank<outT>
{
    T CreateAccount(int sum);
}

classBank<T> : IBank<T>
where T : Account, new()
{
    public T CreateAccount(int sum)
    {
        T acc = new T();
        acc.DoTransfer(sum);
        return acc;
    }
}

classProgram
{
    staticvoid Main(string[] args)
    {
        IBank<DepositAccount> depositBank = new Bank<DepositAccount>();
        Account acc1 = depositBank.CreateAccount(34);

        // Теперь всё работает - это эффект ковариантности
        IBank<Account> ordinaryBank = depositBank;
        Account acc2 = ordinaryBank.CreateAccount(45);
    }
}
```

## Контравариантность интерфейсов

Контравариантность интерфейса, наоборот, позволяет представлять объекты с интерфейсом с менее конкретным типом в виде интерфейсов с более конкретным типом. Допускает использование контравариантных типов только в параметрах методов интерфейса.

```
// Обобщённый интерфейс с контравариантным параметром типа
interface ITransaction<int>
{
    // Метод принимает объект контравариантного типа
    void DoOperation(T account, int sum);
}

// Обобщённый класс, реализующий интерфейс ITransaction для типа T
// С ограничением, что тип T должен быть потомком класса Account
class Transaction<T> : ITransaction<T>
where T : Account
{
    // Реализация метода из интерфейса
    public void DoOperation(T account, int sum)
    {
        account.DoTransfer(sum);
    }
}

static void Main(string[] args)
{
    // Создаём класс Transaction для типа Account
    ITransaction<Account> accTransaction = new Transaction<Account>();
    // И передаём в него объект класса Account
    accTransaction.DoOperation(new Account(), 400);

    // Также создаём класс Transaction для типа Account,
    // Но записываем его в тип интерфейса ITransaction для более конкретного
    типа DepositAccount
    ITransaction<DepositAccount> depAccTransaction = new
    Transaction<Account>();
    // Хотя объект и был создан для типа Account
    // За счёт контравариантности мы можем передавать в его метод объекты
    типа DepositAccount
    depAccTransaction.DoOperation(new DepositAccount(), 450);
}
```

## Обобщенные делегаты

Еще обобщенными могут быть делегаты, о которых пока что нам ничего не известно (см. главу 3).

## Несколько типов-параметров

В принципе, для обобщений можно определять сколько угодно типов-параметров.

Несколько типов параметров перечисляются в треугольных скобках через запятую:

```
class EnormousClass<T, U, V, W>
{
    // ...
}
```

### **Ограничения для нескольких типов-параметров**

Для каждого из типов-параметров можно задавать свои ограничения отдельной секцией `where`:

```
classBase { }

classTest<T, U>
where U : struct
where T : Base, new()
{
    // ...
}
```

### **Пример**

Когда это может быть нужно?

Например, при реализации своего класса словаря, или какой-нибудь структуры в этом духе – коллекций, элементами которых являются пары значений. Или не для коллекций, всё зависит от задачи.

Пример реализации класса словаря приводится в Приложении Е.

### **Ошибки при использовании нескольких параметров типов**

При использовании перегрузок методов в обобщенных классах с несколькими параметрами типов можно допустить ошибку неоднозначности перегрузок, в случае, когда имеется несколько перегрузок методов, отличающихся только типами-параметрами.

```
// Обобщённый класс с двумя типами-параметрами
classGen<T, V>
{
    // Два поля
    private T ob1;
    private V ob2;

    // Метод Set с типом параметра T
    publicvoid Set(T o)
    {
        ob1 = o;
    }

    // Перегрузка метода Set с типом параметра V
    publicvoid Set(V o)
```

```

    {
        ob2 = o;
    }

classAmbiguityExample
{
staticvoid Main()
{
    // Создаём объект класса Gen для типов int и double
    Gen<int, double> ok = new Gen<int, double>();
    // И другой - для типов int и int
    Gen<int, int> notOK = new Gen<int, int>();

    // В случае с первым объектом всё будет работать, потому что понятно
    какой метод вызывать
        ok.Set(10);
    // Здесь будет ошибка из-за неоднозначности. В классе 2 метода с
    одинаковой сигнатурой!
        notOK.Set(10);
    }
}
}

```



*Рисунок 1.1 – Сложно...*

## Глава 2. ОПЕРАТОРЫ

### Задача

Представьте, что нужно реализовать класс для некоторых числовых значений.

Пусть это будут числа некоей воображаемой алгебры некоего Паши, где не существует числа 5, а операции сложения и вычитания не совсем коммутативны.

```
class PInt
{
    private int num;
}
```

Как реализовать операции над ним? Самый очевидный ответ – через методы.

### Класс PInt

Например, вот так:

```
// Класс Числа алгебры Паши (П-числа)
class PInt
{
    // Приватное поле для хранения числа
    private int num;

    // Конструктор
    public PInt(int init)
    {
        num = init;
        // Если число кратно 5, увеличиваем его на единицу
        if (num % 5 == 0)
        {
            ++num;
        }
    }

    // Метод Plus для сложения двух П-чисел
    public PInt Plus(PInt p2)
    {
        // Считаем количество пятерок во втором числе
        int numberOfFives = p2.num / 5;
        // Прибавляем к первому числу второе за исключением количества пятерок в нём
        num += p2.num - numberOfFives;
        // Если число кратно пяти
        if (num % 5 == 0)
        {
            // Увеличиваем на 1
            ++num;
        }
    }
}
```

```

    return this;
}

// Метод Minus для получения разности двух P-чисел
public PInt Minus(PInt p2)
{
    // Считаем количество пятерок во втором числе
    int numberOfFives = p2.num / 5;
    // Отнимаем от первого числа второе за исключением количества пятерок в нём
    num -= p2.num - numberOfFives;
    // Если число кратно пяти
    if (num % 5 == 0)
    {
        // Уменьшаем на 1
        --num;
    }
    return this;
}

// Переопределение метода ToString для вывода числа на консоль с
// суффиксом 'p'
public override string ToString()
{
    return num.ToString() + "p";
}
}

```

## Использование класса PInt

```

static void Main(string[] args)
{
    // Создаём три числа
    PInt p1 = new PInt(245);
    PInt p2 = new PInt(141);
    PInt p3 = new PInt(-300);

    // Выполняем математические операции
    p1 = p1.Plus(p2).Plus(p3);
    p2 = p1.Minus(p3);
}

```

Выглядит очень громоздко. А если еще нужно добавить операции умножения, деления, возведения в степень и т.д.

## Операции

Гораздо лучше было бы работать с нашими объектами, как с обычными числами:

```

// Создавать так:
PInt p1 = 245;
// А не так:
PInt p1 = new PInt(245);

```



```
// Складывать так:
p2 = p1 + p3;
// А не так:
p2 = p1.Plus(p3);

// Вычитать так:
p3 = p1 - p1;
// А не так:
p3 = p1.Minus(p1);
```

И этого можно добиться с использованием перегрузки операторов.

### Перегрузка операторов

Перегрузка операторов – создание логики поведения при использовании стандартных операторов языка над объектами произвольных классов.

Напомним, что операторы в языке обозначаются символами операций (+, -, ==, !=, <, >, ^, | и т.д.).

### Что имеется в виду

```
// Создаём Мишу
Person person1 = new Person("Misha");
// Делим Мишу на 2 и получаем полурослика
Person halfling = person1 / 2;

// Если Миша больше полурослика
if (person1 > halfling)
{
    // Создаём Машу
    Person person2 = new Person("Masha");
    // Складываем Мишу и Машу и получаем ребёнка
    Person child = person1 + person2;

    // Если ребёнок равен Мише
    if (child == person1)
    {
        // Что-то пошло не так...
        Console.WriteLine("Something went wrong...");
    }
}
```

Таблица 8.1

### Операторы в C#

Операторы	Перегружаемость
+, -, !, ~, ++, --, true, false	Можно перегружать
+, -, *, /, %, &,  , <<, >>	Можно перегружать
==, !=, <, >, <=, >=	Можно перегружать, только парами
&&,	Нельзя, но они используют & и
[]	Вместо перегрузки – индексаторы

Операторы	Перегружаемость
()	Нельзя
+=, -=, *=, /=, %=,  =, ^=, <<=, >>=	Перегружаются автоматически при перегрузке соответствующего бинарного оператора
=, .., ?:, ?., ??, ->, =>, f(x), new, is, as, checked, unchecked, default, delegate, sizeof, typeof, nameof	Нельзя перегружать

### Синтаксис

Перегрузка оператора описывается в классе (или структуре), для которого она предназначена, и выглядит практически как описание обычного статического метода:

```
publicstatic возвр_значение operator знак_оператора (аргументы);
```

Вместо имени метода записывается специальное ключевое слово `operator` и знак оператора, который нужно переопределить.

### Аргументы

Минимум один из аргументов оператора должен быть того типа, для которого эта перегрузка создается.

У унарных операторов 1 (один) аргумент, у бинарных – 2 (два) аргумента.

```
class Person
{
    // Перегруженный оператор + для класса Person
    // В качестве аргументов принимает 2 объекта типа Person
    // И возвращает объект типа Person
    publicstatic Person operator +(Person first, Person second)
    {
        // Создаётся новая персона и возвращается
        return new Person();
    }
}

// Класс Игрок
class Player
{
    // Свойство Уровень
    public int Level { get; set; }

    // Перегруженный оператор инкремента ++ с одним параметром типа Player
    // С типом возвращаемого значения Person
    publicstatic Player operator ++(Player p)
```

```

{
// Увеличивает значение свойства Уровень на 1
    p.Level++;
// Возвращает объект, который принял
return p;
}
}

```

## Как это работает

На самом деле, любой оператор в C# – это просто вот такой замаскированный метод.

Если необходимо сложить 2 числа, вы пишете:

```

int a = 2;
int b = 3;
int c = a + b;

```

Но на самом деле это происходит следующим образом (только скрыто от программиста):

```

int c = Int32.operator +(a, b);

```

Вызывается метод соответствующего оператора, а операнды передаются ему в качестве параметров.

## Чуть глубже

На самом деле все еще серьезнее, и выглядит примерно так:

```

int a;
Int32.operator=(a, 2);
int b;
Int32.operator=(b, 3);
int c;
Int32.operator=(c, Int32.operator+(a, b));

```

## Person и Player

Значит, когда мы будем применять операторы для объектов классов, в которых они перегружены, то просто будут вызываться соответствующие оператору методы этого класса.

```

class Person
{
// Перегруженный оператор + для Person и Person в классе Person
public static Person operator +(Person first, Person second)
{
return new Person();
}
}

```

```

class Player
{
public int Level { get; set; }

// Перегруженный оператор ++ в классе Player
public static Player operator ++(Player p)
{
    p.Level++;
    return p;
}
}

class App
{
    static void Main(string[] args)
    {
        Person p1 = new Person();
        // Здесь неявно происходит вызов описанного нами метода operator+
        // Куда p1 передаётся в качестве аргумента first, и p1 в качестве
        // аргумента second
        Person p2 = p1 + p1;
        // То же самое, только в качестве второго аргумента передаётся объект p2
        Person p3 = p1 + p2;

        Player p1 = new Player();
        p1.Level = 1;

        // Неявно вызывается метод operator++, в качестве аргумента передаётся
        // объект p1
        // p1.Level становится равен 2
        ++p1;
        // p1.Level становится равен 3
        p1++;
    }
}

```

## Перегрузка унарных операторов

Унарные операторы можно разделить на 3 группы:

—+, —, !, ~

—++, --

—true, false

Эти группы операторов различаются ограничениями, которые накладываются на их перегрузку.

У всех унарных операторов в качестве параметра должен быть объект типа, в котором они перегружаются.

### Операторы +, —, !, ~

Перегрузки этих операторов могут возвращать любой тип, за исключением void.

```

// Класс комплексного числа
classComplex
{
    // Действительная часть
    privateint a;
    // Мнимая часть
    privateint b;

    public Complex()
    {
        a = b = 0;
    }

    public Complex(int i, int j)
    {
        a = i;
        b = j;
    }

    // Перегрузка оператора унарного минуса, возвращает объект такого же
    // класса Complex
    publicstatic Complex operator -(Complex c)
    {
        // Создаётся новое комплексное число
        Complex temp = new Complex();
        // Меняются знаки его компонент
        temp.a = -c.a;
        temp.b = -c.b;
        // Возвращается инвертированное комплексное число
        return temp;
    }

    // Перегрузка оператора Битового отрицания, возвращаемое значение - int
    publicstaticintoperator ~(Complex c)
    {
        // Возвращает действительную часть числа c
        return c.a;
    }

    // Перегрузка оператора Не, возвращаемое значение - bool
    publicstaticbooloperator !(Complex c)
    {
        // Возвращает true, если мнимая часть числа равна нулю, иначе возвращает
        // false
        return c.b == 0 ? true : false;
    }

    // Заодно переопределён метод ToString
    publicoverridestring ToString()
    {
        // Чтобы в правильной форме отображать комплексное число
        return a + " + " + b + "i";
    }
}
classMyClient

```

```

{
    public static void Main()
    {
        // Создаём комплексное число 10 + 20i
        Complex c1 = new Complex(10, 20);
        Console.WriteLine(c1);
        // Неявно вызываем перегруженный метод оператора унарного минуса для
        // объекта c1.
        // Полученное значение записываем в c2
        Complex c2 = -c1;
        Console.WriteLine(c2);

        // Неявно вызываем перегрузку оператора ~, чтобы получить действительную
        // часть числа c2
        // Если она равна -10
        if (~c2 == -10)
        {
            // Неявно вызываем метод оператора !, чтобы проверить наличие мнимой
            // части
            // Он возвращает true/false и мы можем использовать это в условии if-a
            if (!c2)
            {
                Console.WriteLine("У числа отсутствует мнимая часть");
            }
            else
            {
                Console.WriteLine("Число имеет мнимую часть");
            }
        }
    }
}

```

## Операторы ++, --

Перегрузка оператора инкремента/декремента автоматически распространяется как на постфиксную, так и на префиксную версию оператора.

Метод перегрузки этих операторов обязательно должен возвращать объект того же типа, в котором он объявлен.

## Операторы ++, --

```

// Класс Муравейника
class Anthill
{
    // Свойство Количество муравьёв
    public int AntNumber { get; private set; }

    // Перегрузка оператора инкремента ++
    // Без вариантов: принимает Ants, возвращает Ants
    public static Anthill operator ++(Anthill ants)
    {
        // Увеличиваем количество муравьёв на 1
        ++ants.AntNumber;
    }
}

```

```

// Возвращаем то, что приняли
return ants;
}

// Тоже самое по аналогии
public static Anthill operator --(Anthill ants)
{
    --ants.AntNumber;
    return ants;
}

}

class Program
{
    static void Main(string[] args)
    {
        // Создаём объект муравейника
        Anthill ants = new Anthill();

        // По циклу увеличиваем их количество, пока оно не достигнет 100
        while (ants.AntNumber < 100)
        {
            ants++;
        }
    }
}

```

## Операторы true и false

Во-первых, представьте себе, есть такие операторы.

Во-вторых, они – парные. Это значит, что их можно перегружать только вместе. Если написать один, но не написать другой, компилятор выдаст ошибку.

В-третьих, эти операторы должны всегда возвращать тип bool.

Оператор true вызывается, когда мы пытаемся использовать объект класса в условии if, тернарном операторе или условии цикла.

А когда вызывается оператор false... Об этом чуть позже.

## Оператор true

```

class Anthill
{
    public int AntNumber { get; set; }

    public static Anthill operator --(Anthill ants)
    {
        --ants.AntNumber;
        return ants;
    }

    // Перегрузка оператора true.
    // Принимает Anthill, возвращает bool. Иные типы не допустимы.
}

```

```

publicstaticbooloperatortrue(Anthill ants)
{
    Console.WriteLine("Operator true");
    // Возвращает true, если количество муравьёв больше нуля
    return ants.AntNumber > 0;
}

// По аналогии.
// По своей сути операторы true и false должны быть противоположными
// true проверяет объект на истинность, false - на ложность
publicstaticbooloperatorfalse(Anthill ants)
{
    Console.WriteLine("Operator false");
    // Возвращает true, если количество муравьёв меньше или равно нулю
    return ants.AntNumber <= 0;
}

}

classProgram
{
    staticvoid Main(string[] args)
    {
        Anthill anthill = new Anthill();
        anthill.AntNumber = 0;
        // Вот здесь вызовется оператор true
        // Поскольку количество муравьёв не больше нуля, метод оператора вернёт
        false
        if (anthill)
        {
            Console.WriteLine("Anthill inhabited");
        }
        else
        {
            // Следовательно, выполнится этот блок else
            Console.WriteLine("Anthill abandoned");
        }
        anthill.AntNumber = 3;
        // Здесь оператор true будет выполняться по циклу,
        // Пока не вернёт false и цикл не закончится
        while (anthill)
        {
            anthill--;
            Console.WriteLine("Ants left: {0}", anthill.AntNumber);
        }
    }
}

```

## Бинарные операторы

Бинарные операторы называются бинарными, потому что применяются к двум операндам, и, значит, принимают два параметра.

Поскольку мы описываем перегрузку для какого-либо конкретного класса, хотя бы один из двух параметров должен быть этого типа.



Тип возвращаемого значения – любой, кроме void.

После перегрузки оператора автоматически появляется возможность использовать операторы смешанного присваивания (оператор плюс присваивание). Т.е. если вы перегрузили оператор бинарного +, то сможете использовать оператор +=.

### **Второй аргумент**

Второй аргумент – самое интересное, что есть в перегрузке бинарных операторов.

Дело в том, что второй аргумент может быть любого типа.

А это позволяет делать перегрузки перегрузок операторов.

Рассмотрим на примере бинарного оператора + (оператора сложения). Мы можем определить в классе сколько нам потребуется перегрузок этого оператора, для разных типов второго аргумента.

Пусть будет возможность складывать объект нашего класса с числом, объект со строкой, два объекта друг с другом – любые варианты комбинаций.

Порядок аргументов у оператора имеет значение. Поэтому, обычно, если вы перегружаете «Класс + число», то следует и перегрузить «Число + класс».

### **Числа алгебры Паши**

Перепишем класс П-чисел, заменив в нем методы для сложения и вычитания на соответствующие операторы:

```
// Всё тот же класс П-чисел
class PInt
{
    private int num;

    public PInt(int init)
    {
        num = init;
        if (num % 5 == 0)
        {
            ++num;
        }
    }

    public PInt(PInt p)
    {
        num = p.num;
    }

    // Вместо метода Plus - перегрузка оператора сложения
    // Возвращает PInt и принимает 2 значения типа PInt
    public static PInt operator +(PInt left, PInt right)
    {
        int numberOfFives = right.num / 5;
        PInt retP = new PInt(left.num);
    }
}
```

```

        retP.num += right.num - numberOfFives;
    if (retP.num % 5 == 0)
    {
        ++retP.num;
    }
    return retP;
}

// Перегрузка оператора вычитания
// Также возвращает PInt и принимает 2 значения PInt
public static PInt operator -(PInt left, PInt right)
{
    int numberOfFives = right.num / 5;
    PInt retP = new PInt(left.num);
    retP.num -= right.num - numberOfFives;
    if (retP.num % 5 == 0)
    {
        --retP.num;
    }
    return retP;
}

public override string ToString()
{
    return num.ToString() + "p";
}
}

class Program
{
    static void Main(string[] args)
    {
        PInt p1 = new PInt(245);
        PInt p2 = new PInt(141);
        PInt p3 = new PInt(-300);

        Console.WriteLine("{0} + {1} = {2}", p1, p2, p1 + p2);
        Console.WriteLine("{0} + {1} = {2}", p2, p1, p2 + p1);

        // Теперь можно складывать и вычитать PInt, как обычные числа
        p1 = p1 + p2 - p3;
        p2 = p1 + p3;
    }
}

```

## Продолжение

А теперь добавим возможность прибавлять к объекту PInt обычные числа типа int:

```

// [Внутри класса]
// Перегрузка оператора +, который принимает в качестве аргументов
// объект PInt и целое число

```

```

publicstatic PInt operator +(PInt left, int right)
{
    if (right % 5 != 0)
    {
        returnnew PInt(left.num + right);
    }
    else
    {
        returnnew PInt(0);
    }
}

// [B Main]
// Теперь такой код будет работать:
PInt p4 = p1 + 14;
// Но такой - вызовет ошибку:
PInt p5 = 14 + p1;

```

## Продолжение 2

Чтобы операция работала в обе стороны, нужно добавить аналогичный метод с обратным порядком аргументов:

```

// [Внутри класса]
// Перегрузка PInt + int
publicstatic PInt operator +(PInt left, int right)
{
    if (right % 5 != 0)
    {
        returnnew PInt(left.num + right);
    }
    else
    {
        returnnew PInt(0);
    }
}
// Перегрузка int + PInt
publicstatic PInt operator +(int left, PInt right)
{
    if (left % 5 != 0)
    {
        returnnew PInt(left + right.num);
    }
    else
    {
        returnnew PInt(0);
    }
}

// [B Main]
// Оба варианта рабочие:
PInt p4 = p1 + 14;
PInt p5 = 14 + p1;

```

### Продолжение 3

Добавим еще 2 перегрузки для оператора умножения:

```
[Внутри класса]
// Перегруженный оператор умножения *
// Для типов PInt и char
// Возвращает строку
public static string operator *(PInt left, char right)
{
    // Возвращает строковое представление П-числа с добавочным символом,
    // переданным вторым параметром
    return left.ToString().Replace('p', right);
}
// Ещё один перегруженный оператор умножения *
// Для типов int и PInt
// Возвращает int
public static int operator *(int left, PInt right)
{
    // Возвращает обычное число, равно произведению чисел из параметров
    return left * right.num;
}
// Конечно, для них желательно добавить обратные перегрузки (char * PInt
// и PInt * int)

// [В Main]
// Теперь мы можем делать так:
Console.WriteLine(p4);
Console.WriteLine(p4 * 'П');
// И так:
int num = 4 * p4 + 12;
```

### Операторы & и |

Сами по себе эти операторы не представляют ничего интересного, интересно то, где они используются.

#### Срываем маски

Помните операторы логического объединения И (&&) и ИЛИ (||)? Они не просто так похожи на своих побитовых собратьев.

На самом деле, оператор логического И (&&) это просто замаскированная запись:

```
a && b
// эквивалентно
operator false(a) ? a : (a & b);
```

А оператор логического ИЛИ (||):

```
a || b
// эквивалентно
operator true(a) ? a : (a | b)
```

В C# это единственное место, в котором вызывается оператор false.

## Операторы && и ||

Это значит, что, если вы хотите использовать для объектов вашего класса операторы логического объединения && и ||, следует перегрузить операторы & и |, плюс операторы true и false.

Причем методы перегрузки операторов & и | должны обязательно принимать в качестве параметров два объекта вашего класса, и возвращать объект того же типа.

### Демонстрация

```
// Класс муравейника
class Anthill
{
    // Свойство Количество муравьёв
    public int AntNumber { get; set; }

    // Конструктор для задания начального количества муравьёв
    public Anthill(int initAntsNum)
    {
        AntNumber = initAntsNum;
    }

    // Перегрузка оператора true
    // Проверка объекта Муравейника на истинность
    public static bool operator true(Anthill ants)
    {
        Console.WriteLine("Operator true");
        // Муравейник истинный, когда количество муравьёв в нём больше 0
        return ants.AntNumber > 0;
    }

    // Перегрузка оператора false
    // Проверка объекта Муравейника на ложность
    public static bool operator false(Anthill ants)
    {
        Console.WriteLine("Operator false");
        // Муравейник ложный, когда количество муравьёв в нём меньше или равно 0
        return ants.AntNumber <= 0;
    }

    // Перегрузка оператора &
    // Чтобы с ней работал оператора &&, метод должен
    // Принимать 2 объекта Anthill и возвращать объект Anthill
    public static Anthill operator &(Anthill ants1, Anthill ants2)
    {
        Console.WriteLine("Operator &");
        // Возвращает муравейник с меньшим количеством муравьёв (из 2
        // полученных)
        return ants1.AntNumber < ants2.AntNumber ? ants1 : ants2;
    }

    // Перегрузка оператора |
    public static Anthill operator |(Anthill ants1, Anthill ants2)
```

```

{
    Console.WriteLine("Operator |");
    // Возвращает муравейник с Большим количеством муравьёв
    return ants1.AntNumber > ants2.AntNumber ? ants1 : ants2;
}

}

class Program
{
    static void Main(string[] args)
    {
        // Создаём 2 муравейника
        // Первый - с 13 муравьями
        Anthill a1 = new Anthill(13);
        // Второй - с 0
        Anthill a2 = new Anthill(0);

        // В условии if-а выполняем оператор логического И &&
        // Это будет вызывать конструкцию следующего вида:
        // Anthill.operator false(a1) ? a1 : (a1 & a2)
        // Поскольку эта операция записана в условии if-а, то полный её вид
        // будет таким:
        // Anthill.operator true(Anthill.operator false(a1) ? a1 : (a1 & a2))
        if (a1 && a2)
        {
            Console.WriteLine("a1 && a2 == true\n");
            Console.WriteLine("Both anthill are inhabited");
        }
        Console.WriteLine();

        // По аналогии
        // a1 || a2 эквивалентно Anthill.operator true(a1) ? a1 : (a1 | a2)
        // Целиком: Anthill.operator true(Anthill.operator true(a1) ? a1 : (a1 |
a2))
        if (a1 || a2)
        {
            Console.WriteLine("a1 || a2 == true");
            Console.WriteLine("One of the anthills are inhabited");
        }
    }
}

```

## Логические операторы

Все логические операторы – парные, т.е. перегружаются только парами по два противоположных оператора.

– < и >

– <= и >=

– == и !=

Должны принимать хотя бы один объект вашего класса, а возвращать объект булевого типа.

## Больше / меньше

```
// Класс Комната
classRoom
{
    // Свойства Длина, Ширина и Высота стен
    publicfloat Length { get; set; }
    publicfloat Width { get; set; }
    publicfloat WallHeight { get; set; }

    // Конструктор для задания значений свойств
    // У высоты стен значение по умолчанию - 3 метра
    public Room(float length, float width, float height = 3.0f)
    {
        Length = length;
        Width = width;
        WallHeight = height;
    }

    // Перегрузка пары операторов больше-меньше (> и <)
    // Принимают в качестве параметров 2 объекта Room
    publicstaticbooloperator>(Room room1, Room room2)
    {
        // Сравнивает через оператор > площади этих комнат и возвращает
        результат сравнения
        return (room1.Length * room1.Width) > (room2.Length * room2.Width);
    }
    publicstaticbooloperator<(Room room1, Room room2)
    {
        // Сравнивает через оператор < площади этих комнат и возвращает
        результат сравнения
        return (room1.Length * room1.Width) < (room2.Length * room2.Width);
    }

    // Ещё одна перегрузка пары операторов больше-меньше
    // Только теперь для типов параметров Room и float
    publicstaticbooloperator>(Room room, float square)
    {
        // Сравнивает площадь переданной комнаты с заданной площадью
        return (room.Length * room.Width) > square;
    }
    publicstaticbooloperator<(Room room, float square)
    {
        return (room.Length * room.Width) < square;
    }
}

classProgram
{
    staticvoid Main(string[] args)
    {
        // Создаём две комнаты
```

```

Room kitchen = new Room(2.6f, 3.2f);
Room bedroom = new Room(4.2f, 3.9f);

// Сравнение: если одна комната больше другой
// По нашей перегрузке комнаты будут сравниваться по их площадям
if (bedroom > kitchen)
{
    Console.WriteLine("The bedroom is larger than kitchen.");
}

// Комната сравнивается с числом 6
// Вызывается другая перегрузка оператора, для Room и float
// Площадь комнаты kitchen сравнивается с 6
if (kitchen < 6)
{
    Console.WriteLine("The kitchen is small");
}
// Проверяется, попадает ли площадь комнаты в диапазон (6;16)
elseif (kitchen > 6 && kitchen < 16)
{
    Console.WriteLine("The kitchen is normal size");
}
else
{
    Console.WriteLine("The kitchen is big");
}
}
}

```

### Проверка на равенство

При перегрузке операторов == и != рекомендуется также переопределять метод Equals (object). Иначе выйдет так, что проверка на равенство разными вариантами будет возвращать разные результаты, а это, согласитесь, выглядит странно и может привести к ошибкам.

А переопределяя метод Object Equals(object) нужно также переопределить метод GetHashCode (в котором в принципе можно ничего нового не писать).

Значит, если вы хотите определить операции сравнения для своего класса, в нем нужно дописать: оператор ==, оператор !=, метод Equals (object) и метод GetHashCode().

### Пример

```

classRoom
{
    publicfloat Length { get; set; }
    publicfloat Width { get; set; }
    publicfloat WallHeight { get; set; }

    public Room(float length, float width, float height = 3.0f)

```



```

    {
        Length = length;
        Width = width;
        WallHeight = height;
    }

    // Переопределение метода Equals(object)
    public override bool Equals(object obj)
    {
        // Если переданный объект является комнатой
        if (obj is Room room)
        {
            // Производим сравнение по вычисленной площади
            // Сравнивается комната this и переданная в качестве параметра
            return (this.Length * this.Width) == (room.Length * room.Width);
        }
        else
        {
            // Если второй объект не комната, то они точно не равны - возвращаем
            false
            return false;
        }
    }

    // Переопределение метода GetHashCode()
    public override int GetHashCode()
    {
        // Просто возвращаем результат вызова родительского метода GetHashCode
        return base.GetHashCode();
    }

    // Перегрузка оператора равенства == для двух комнат
    public static bool operator ==(Room room1, Room room2)
    {
        // Вызывает метод Equals для первой комнаты, передавая вторую в качестве
        параметра
        // Возвращает результат, который вернёт Equals
        return room1.Equals(room2);
    }

    // Перегрузка оператора проверки на неравенство != двух комнат
    public static bool operator !=(Room room1, Room room2)
    {
        // То же самое, что проверка на равенство, только обратное (с помощью
        оператора !=)
        return !room1.Equals(room2);
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Создаём 2 комнаты

```

```

        Room kitchen = new Room(2.6f, 3.2f);
        Room bedroom = new Room(3.2f, 2.6f);

        Console.WriteLine("(1): ");
        // Проверяем эти комнаты на равенство
        // Т.е. проверяется равенство их площадей
        if (kitchen == bedroom)
        {
            Console.WriteLine("The kitchen and bedroom are equal in
size");
        }
        // Проверяем их на неравенство
        if (kitchen != bedroom)
        {
            Console.WriteLine("The kitchen and bedroom are not equal in
size");
        }

        Console.WriteLine("(2): ");
        // Проверка на равенство при помощи метода Equals
        if (kitchen.Equals(bedroom))
        {
            Console.WriteLine("The kitchen and bedroom are equal in
size");
        }
        // Проверка на неравенство при помощи Equals
        if (!kitchen.Equals(bedroom))
        {
            Console.WriteLine("The kitchen and bedroom are not equal in
size");
        }
    }
}

```

## Преобразования типов

Вспомним про стандартный механизм преобразования типов.

Преобразования бывают явными и неявными.

Для явного преобразования необходимо перед переменной написать в круглых скобках желаемый тип:

```

int a = 2;
// явное преобразование int в char
char c = (char)a;

```

Для неявного преобразования ничего указывать не нужно, оно происходит само на основании типов:

```

int a = 2;
// неявное преобразование int в double
double d = a;

```

## Операторы преобразования типов

Так вот, все подобные преобразования выполняются с помощью вызова специальных операторов преобразования типов.

И эти операторы бывают двух видов:

- `explicit` - для явного преобразования;
- `implicit` - для неявного преобразования.

### Синтаксис

Методы перегрузки операторов преобразования описываются следующим образом:

```
public static [explicit/implicit] operator ToType(fromType param)
{
    // ...
}
```

1. Перед словом `operator` идет ключевое слово `explicit` или `implicit`, в зависимости от того, явное или неявное преобразование мы хотим разрешить.

2. Параметр – объект, который будет преобразовываться. Тип возвращаемого значения – тип, к которому объект из параметра будет преобразован.

3. Тип возвращаемого значения описывается здесь не на своем привычном месте: он должен записываться после слова `operator`, а не перед, как в обычных методах.

4. Один из типов (принимаемый/возвращаемый) должен быть типом, в котором этот оператор описывается.

Т.е. можно описывать как операторы преобразования из какого-то типа в ваш, так и, наоборот – из вашего типа в какой-либо.

## Пример

```
// Класс для примера
class SampleClass
{
    // Целочисленное свойство для примера
    public int SampleProperty { get; set; }

    // Перегрузка оператора явного преобразования типа SampleClass к типу int
    public static explicit operator int(SampleClass sc)
    {
        // Возвращает значение из целочисленного свойства SampleProperty объекта sc
        return sc.SampleProperty;
    }
}

class MainClass
```

```

{
staticvoid Main(string[] args)
{
// Создаётся объект класса SampleClass
var obj = new SampleClass();
obj.SampleProperty = 33;
// Явное преобразования этого объекта к типу int
int a = (int)obj;
// a == 33
}
}

```

## Оператор неявного преобразования

```

classPerson
{
// Поле и свойство Имя
privatestring name;
publicstring Name
{
get
{
// Если значение - пустая строка, возвращаем строку "Неизвестно"
return name == "" ? "Unknown" : name;
}
set
{
name = value;
}
}
// Поле и свойство Фамилия
privatestring surname;
publicstring Surname
{
get
{
// Если фамилия не задана - возвращаем строку "Неизвестно"
return surname == "" ? "Unknown" : surname;
}
set
{
surname = value;
}
}
// Поле и свойство Возраст
// Обратите внимание, что типа string
// Это нужно для того, чтобы возвращать строку "Неизвестно", если
возраст не задан
privatestring age;
publicstring Age
{
get
{
// Если возраст меньше нуля, возвращаем строку "Неизвестно"

```

```

    // *Чтобы сравнить возраст с нулем (число), нужно преобразовать строку к
числу
    return int.Parse(age) < 0 ? "Unknown" : age;
}
set
{
    // Проверка, находится ли в строке value числовое значение
    // Через попытку преобразования value к числу
    if (int.TryParse(value, out int a))
    {
        // Если преобразовать получилось - значит эта строка представляет собой
числовое значение
        // Всё нормально, и можно строку записать в age
        age = value;
    }
}

// Конструктор по умолчанию
public Person()
{
    Name = "";
    Surname = "";
    // Задаёт возраст равным -1, чтобы выдавалась строка "Неизвестно"
    Age = "-1";
}

// Перегрузка оператора явного преобразования типа string в тип Person
public static explicit operator Person(string str)
{
    // Создаём новый объект, который в итоге будем возвращать
    Person ret = new Person();
    // Разбиваем входную строку на подстроки с пробелом в качестве
разделителя
    string[] strs = str.Split(' ');
    // Если количество получившихся подстрок больше 2 (как минимум 3)
    // Иначе условие читается, как "Если есть строка под индексом 2"
    if (strs.Length > 2)
    {
        // Записываем строку под индексом 2 в свойство Age возвращаемого объекта ret
        ret.Age = str[2];
    }
    // Если есть строка под индексом 1
    if (strs.Length > 1)
    {
        // Записываем эту строку в свойство Surname
        ret.Surname = str[1];
    }
    // Если есть строка под индексом 0
    if (strs.Length > 0)
    {
        // Записываем её в свойство Name
        ret.Name = str[0];
    }
}

```

```

// Возвращаем созданный объект
return ret;
}

// Оператор явного преобразования типа Person к типу int
public static explicit operator int(Person p)
{
    // Возвращаем преобразованное к числу значение возраста персоны p
    return int.Parse(p.age);
}

// Методы для быстрого вывода на консоль информации о персоне
public void Show()
{
    Console.WriteLine("Person {0} {1}, age: {2}", Name, Surname, Age);
}

class Program
{
    static void Main(string[] args)
    {
        // Явно преобразуем строку к типу Person и записываем в pers1
        // Здесь вызовется метод оператора explicit operator Person(string str)
        // Куда строка "John Smith" передастся в качестве аргумента str
        Person pers1 = (Person)"John Smith";
        // Тоже самое
        Person pers2 = (Person)"Dumbo";
        // Тоже самое
        Person pers3 = (Person)"Scott Fitzgerald 44";
        // Теперь можно посмотреть, что у нас получилось
        pers1.Show();
        pers2.Show();
        pers3.Show();

        // Явное преобразование объекта Person к типу int
        int pers3age = (int)pers3;
        // Должно вывести возраст третьей персоны
        Console.WriteLine(pers3age);
    }
}

```

## Неявное преобразование

Заменяв слово `explicit` на `implicit` в предыдущем примере, можно посмотреть на неявное преобразование. Тогда код будет выглядеть следующим образом:

```

static void Main(string[] args)
{
    // Теперь не нужно указывать тип Person в скобках
    // Преобразование выполняется неявно
    Person pers1 = "John Smith";
    Person pers2 = "Dumbo";
}

```

```

        Person pers3 = "Scott Fitzgerald 44";
        pers1.Show();
        pers2.Show();
        pers3.Show();

// Как и здесь
int pers3age = pers3;
Console.WriteLine(pers3age);
}

```

## Применение

Может возникнуть вопрос, зачем тогда вообще явные преобразования, если неявные гораздо удобнее?

Однако все совершенно наоборот – рекомендуется использовать только явные преобразования, а неявные применять лишь в случае крайней необходимости.

## Почему?

Потому что неявные преобразования происходят скрыто от программиста.

Их труднее найти в коде, да о них просто можно и не знать. С неявным преобразованием нельзя понять – было ли оно использовано умышленно, или программист вообще не подозревал, что там есть преобразование.

## Финальный пример с числами Паши

```

// Класс П-числа
class PInt
{
// Приватное поле для хранения внутреннего значения
private int num;

// Конструктор на основании целого числа
public PInt(int init)
{
    num = init;
    if (num % 5 == 0)
    {
        ++num;
    }
}

// Конструктор копирования
public PInt(PInt p)
{
    num = p.num;
}

// Оператор сложения двух П-чисел
public static PInt operator +(PInt left, PInt right)
{
    int numberOfFives = right.num / 5;
    PInt retP = new PInt(left.num);
}
}

```

```

        retP.num += right.num - numberOfFives;
    if (retP.num % 5 == 0)
    {
        ++retP.num;
    }
    return retP;
}

// Оператор вычитания двух П-чисел
public static PInt operator -(PInt left, PInt right)
{
    int numberOfFives = right.num / 5;
    PInt retP = new PInt(left.num);
    retP.num -= right.num - numberOfFives;
    if (retP.num % 5 == 0)
    {
        --retP.num;
    }
    return retP;
}

// Оператор умножения двух П-чисел
public static PInt operator *(PInt left, PInt right)
{
    PInt retP = new PInt(left.num * right.num);
    return retP;
}

// Оператор деления двух П-чисел
public static PInt operator /(PInt left, PInt right)
{
    PInt retP = new PInt(((left.num / right.num) + (right.num /
left.num)) / 2);
    return retP;
}

// Оператор неявного преобразования int к PInt
public static implicit operator PInt(int arg)
{
    // Создаёт и возвращает объект нового П-числа, на основе переданного
    параметра
    return new PInt(arg);
}

// Оператор явного преобразования PInt к int
public static explicit operator int(PInt arg)
{
    // Просто возвращает значение поля num
    return arg.num;
}

// Переопределение метода ToString
public override string ToString()
{

```



```

return num.ToString() + "p";
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Благодаря перегрузке оператора преобразования int к PInt
        // Можно создавать числа PInt так просто:
        PInt p1 = 245;
            PInt p2 = 141;
            PInt p3 = -300;

        // Благодаря перегрузкам математических операторов
        // Можно описывать операции вот так:
        PInt p4 = p1 + p2;
            PInt p5 = p1 - p2;
            PInt p2 *= p5;
            PInt p7 = p1 / p6;

        // Явное преобразование к int-y
        int num = (int)p7;

    }
}

```

## Глава 3. ДЕЛЕГАТЫ, ЛЯМБДЫ И СОБЫТИЯ

### Делегаты

Делегаты – объекты, которые указывают на методы.

С помощью объекта типа делегат можно вызывать хранимые в нем методы.

### Описание делегата

Как классы, интерфейсы, структуры и перечисления, делегаты являются специальными типами объектов, и требуют предварительного описания перед использованием.

Делегаты описываются с помощью ключевого слова `delegate`:

```
delegate возвращ_тип ИмяДелегата(список_параметров);
```

Описание делегата похоже на объявление абстрактного метода без реализации.

### Характеристики делегатов

Тип делегата характеризуется его типом возвращаемого значения.

Делегат может ссылаться только на методы, которые соответствуют его типу возвращаемого значения и типам параметров.

## Объяснение

Объект вот такого делегата `FunctionRef`:

```
delegate void FunctionRef();
```

сможет хранить только методы, которые возвращают `void` и которые не имеют параметров.

А объект делегата `MyDelegate`:

```
delegate float MyDelegate(Dog a, Dog b);
```

сможет хранить только методы, которые возвращают `float` и принимают два объекта типа `Dog`.

## Суть

В делегат можно записать ссылку на любой метод, который соответствует его описанию.

Это может быть как статический метод класса, так и метод некоторого объекта.

Самое главное, что метод, записанный в делегате, можно вызывать. А поскольку делегат является переменной, метод в нем можно перезаписывать и менять.

Это дает возможность определять вызываемый метод не во время компиляции, а прямо во время выполнения.

## Использование делегата

Делегат создается, как и любой объект, через описание типа и имени.

Допустим, вы описали следующий тип делегата:

```
delegate void DelegateType(int a);
```

Тогда, чтобы создать объект этого делегата, нужно просто написать:

```
static void Main(string[] args)
{
    DelegateType d;
}
```

## Инициализация делегата

При создании у делегата нет значения (`null`). Значениями для делегата служат методы.

Пусть у нас есть следующий метод `ShowInt`:

```
public static void ShowInt(int num)
{
    Console.WriteLine("Переданное число: {0}", num);
    Console.WriteLine("Его квадрат: {0}", num);
}
```

Тогда можно присвоить нашему делегату этот метод:

```
DelegateType d;  
d = ShowInt;
```

И вызывать:

```
d(1234);
```

## Пример

```
// Статический класс для статических методов  
staticclass MyMath  
{  
    // Обратите внимание, что у всех методов одинаковые  
    // типы возвращаемого значения и типы параметров  
    public static int Sum(int a, int b)  
    {  
        return a + b;  
    }  
  
    public static int Sub(int a, int b)  
    {  
        return a - b;  
    }  
  
    public static int Mul(int a, int b)  
    {  
        return a * b;  
    }  
  
    public static int Div(int a, int b)  
    {  
        return a / b;  
    }  
}  
  
// Определён тип делегата с возвращаемым значением типа int  
// И двумя параметрами типа int  
delegate int Operation(int arg1, int arg2);  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        // Создаём 2 целочисленных переменных  
        int x = 3, y = 4;  
        // Создаём делегат  
        Operation op;  
  
        // Назначаем ему статический метод Sum из класса MyMath  
        op = MyMath.Sum;  
        // Вызываем делегат, передавая ему в качестве параметров x и y
```

```

// Результат сохраняем в переменную res
// Здесь произойдёт вызов метода MyMath.Sum
int res = op(x, y);
// Вывод результата
    Console.WriteLine("1) {0} + {1} = {2}", x, y, res);

// Далее назначаем делегату метод Sub
    op = MyMath.Sub;
// И вызываем делегат
// Теперь, поскольку в нём уже метод Sub, произойдёт вызов именно метода
Sub
res = op(x, y);
    Console.WriteLine("2) {0} - {1} = {2}", x, y, res);

// Назначаем делегату метод Mul
    op = MyMath.Mul;
// Вызываем делегат == вызываем метод Mul
    res = op(x, y);
Console.WriteLine("3) {0} * {1} = {2}", x, y, res);

// Назначаем делегату метод Div
    op = MyMath.Div;
// Вызываем делегат == вызываем метод Div
    res = op(x, y);
Console.WriteLine("4) {0} / {1} = {2}", x, y, res);
}
}

```

## Еще пример

```

classProgram
{
// Объявляем тип делегата:
// Без возвращаемого значения и без параметров
delegate void Message();

// И 2 метода для него:
// Один выводит на консоль "Доброе утро"
private static void GoodMorning()
{
    Console.WriteLine("Good Morning");
}

// Другой выводит на консоль "Добрый вечер"
private static void GoodEvening()
{
    Console.WriteLine("Good Evening");
}

static void Main(string[] args)
{
// Создаём делегат
    Message mes;
// Если сейчас по времени меньше 12 часов

```

```

if (DateTime.Now.Hour < 12)
{
    // Назначаем делегату метод GoodMorning
    mes = GoodMorning;
}
else
{
    // Иначе назначаем делегату метод GoodEvening
    mes = GoodEvening;
}
// Вызываем делегат
// В итоге выполнится либо метод GoodMorning, либо GoodEvening - в
зависимости от времени
mes();
Console.ReadKey(true);
}
}

```

### Множественное назначение

Есть еще одна уникальная черта делегата:

Делегату может быть назначен не один метод, а любое количество. Это позволяет создавать так называемую цепочку вызовов: при вызове делегата методы, назначенные ему, последовательно вызываются друг за другом.

Для добавления к делегату дополнительных методов используются операторы + и +=.

А для удаления делегата применяются операторы - и -=.

### Демонстрация

```

// Методы для работы с массивом
// Все методы возвращают void и принимают массив
static class ArrayUtils
{
    // Обобщённый метод для вывода массива на консоль
    public static void ShowArray<T>(T[] arr)
    {
        Console.Write("Array: ");
        // С помощью метода Join класса string
        // Массив преобразовывается в строку с добавлением запятых между
элементами
        Console.WriteLine(string.Join(", ", arr));
    }

    // Метод для вывода максимального элемента массива
    public static void ShowMax(int[] arr)
    {
        if (arr.Length == 0)
        {
            Console.WriteLine("Array is empty");
        }
        int max = arr[0];
        for (int i = 1; i < arr.Length; ++i)

```

```

        {
            if (max < arr[i])
            {
                max = arr[i];
            }
        }
        Console.WriteLine("Max value in the array is {0}", max);
    }

    // Метод для вывода минимального элемента массива
    public static void ShowMin(int[] arr)
    {
        if (arr.Length == 0)
        {
            Console.WriteLine("Array is empty");
            return;
        }
        int min = arr[0];
        for (int i = 1; i < arr.Length; ++i)
        {
            if (min > arr[i])
            {
                min = arr[i];
            }
        }
        Console.WriteLine("Min value in the array is {0}", min);
    }

    // Метод для вывода среднего значения элементов массива
    public static void ShowAvg(int[] arr)
    {
        if (arr.Length == 0)
        {
            Console.WriteLine("Array is empty");
            return;
        }
        int sum = 0;
        for (int i = 0; i < arr.Length; ++i)
        {
            sum += arr[i];
        }
        float avg = (float)sum / arr.Length;
        Console.WriteLine("Average value in the array is {0}", avg);
    }

    // Метод для вывода нормализованного массива
    // *Нормализованный - все элементы редуцированы до значений -1, 0 и 1
    public static void ShowNormalize(int[] arr)
    {
        int[] tmpArr = new int[arr.Length];
        arr.CopyTo(tmpArr, 0);
        for (int i = 0; i < tmpArr.Length; ++i)
        {
            // Вс положительные элементы
            if (tmpArr[i] > 0)

```

```

        {
// Заменяются на 1
            tmpArr[i] = 1;
        }
// Все отрицательные элементы
elseif (tmpArr[i] < 0)
    {
// Заменяются на -1
        tmpArr[i] = -1;
    }
}
Console.Write("Normalize array: ");
Console.WriteLine(string.Join(", ", tmpArr));
}
}

// Объявлен тип делегата для вышеописанных методов
delegate void MultiDelegate(int[] array);

class Program
{
static void Main(string[] args)
{
// Создаётся массив
int[] array1 = { -2, 17, 23, 6, -5, 14, 0, -11, -2 };

// Создаётся делегат, в него записывается метод ShowArray
// *Конкретный тип для обобщённого метода определяется автоматически
// На основании типов делегата
    MultiDelegate multigate = ArrayUtils.ShowArray;
// А затем к делегату добавляются другие методы
    multigate += ArrayUtils.ShowAvg;
// В том порядке, в котором они добавляются
    multigate += ArrayUtils.ShowMax;
// Они и будут вызываться
    multigate += ArrayUtils.ShowMin;
    multigate += ArrayUtils.ShowNormalize;

// Вызов делегата запускает цепочку вызовов
// Всех записанных в него методов
    multigate(array1);
    Console.WriteLine();

// Из делегата удаляются некоторые методы
    multigate -= ArrayUtils.ShowMin;
    multigate -= ArrayUtils.ShowMax;
    multigate -= ArrayUtils.ShowAvg;

// Делегат вызывается снова
// Теперь будут выполнены лишь оставшиеся методы
    multigate(array1);
}
}

```

## Ковариантность и контравариантность делегатов

У делегатов также имеются эти «страшные» свойства. Только, в отличие от обобщенных интерфейсов, для делегатов они всегда действуют по умолчанию.

Свойство ковариантности позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата.

Свойство контравариантности позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

### Пример

```
// Класс Здание
class Building
{
    // ...
}

// Производный от него класс Школа
class School : Building
{
    // ...
}

// Производный от здания класс Отель
class Hotel : Building
{
    // ...
}

// Класс строительной компании
class ConstructCompany
{
    // Метод Постройки школы, возвращающий новый объект школы
    public School BuildSchool()
    {
        Console.WriteLine("Construction company is building a school");
        // Создаётся и возвращается объект класса School
        return new School();
    }

    // Метод Постройки отеля, возвращающий объект отеля
    public Hotel BuildHotel()
    {
        Console.WriteLine("Construction company is building a new
hotel");
        // Создаётся и возвращается объект Hotel
        return new Hotel();
    }
}

// Класс Горожанин
```



```

class Citizen
{
    // Метод Посещения постройки, принимает постройку в качестве параметра
    public void VisitBuilding(Building someBuilding)
    {
        Console.WriteLine("The citizen decided to visit the Building");
    }

    // Методы Посещения отеля, принимает объект класса Hotel
    public void VisitHotel(Hotel hotel)
    {
        Console.WriteLine("The citizen decided to visit the Hotel");
    }
}

// Делегат для демонстрации ковариантности
// Возвращаемым типом указан Building
delegate Building CreateBuilding();

// Делегат для демонстрации контравариантности
// Принимает аргумент типа Hotel
delegate void Visit(Hotel building);

class Program
{
    static void Main(string[] args)
    {
        // Создаём строительную компанию
        ConstructCompany company = new ConstructCompany();

        // Создаём делегата типа CreateBuilding
        CreateBuilding build;
        Random randGen = new Random();
        int randNum = randGen.Next(2);
        // 50 на 50
        if (randNum == 0)
        {
            // Что назначим делегату метод BuildSchool
            // Хотя у делегата указано, что тип возвращаемого значения должен быть
            Building
            // Мы можем назначить метод, который возвращает School - производный от
            // указанного класса Building
            // Это и есть ковариантность
            build = company.BuildSchool;
        }
        else
        {
            // Или метод BuildHotel
            build = company.BuildHotel;
        }
        // Вызываем делегата и сохраняем полученную постройку в переменную b
        Building b = build();

        // Создаём горожанина
    }
}

```

```

        Citizen adler = new Citizen();
    // Создаём делегат типа Visit и назначаем ему метод VisitBuilding
    // Хотя у делегата отмечено, что он должен принимать параметр типа Hotel
    // Мы можем назначит ему метод, принимающий параметр типа Building -
    базовый для класса Hotel
    // Это - контравариантность
    Visit visit = adler.VisitBuilding;
    // А потом ещё раз (можно несколько раз назначать делегату один метод)
    // Тогда он просто вызовется несколько раз
    visit += adler.VisitBuilding;
    // Также добавляем к делегату метод VisitHotel,
    visit += adler.VisitHotel;
    // Иещёодин VisitBuilding
    visit += adler.VisitBuilding;

    // Если строительная компания построила отель, а не школу
    if (b is Hotel hot)
    {
        // Вызываем делегат, и передаём в качестве параметра объект отеля hot
        // Полученный путём преобразования оператора is в условии проверки if
        visit(hot);
    }
}

```

## Делегаты и методы объекта

Делегат спокойно может указывать на метод некоторого объекта:

```

// КлассАвиакомпания
classAirline
{
    privatestatic Random gen;

    static Airline()
    {
        gen = new Random();
    }

    // Метод Отправить самолёт в рейс
    publicint SendFlight()
    {
        // Случайное количество пассажиров
        int passengers = gen.Next(140, 780);
        // Выводим на консоль, что, мол, столько-то человек отправилось в
        небесное плавание
        Console.WriteLine("The plane leaves with {0} passengers on board",
        passengers);
        // Возвращаем количество отправленных пассажиров
        return passengers;
    }

    // Метод задержать самолёт
    publicint DelayedFlight()

```

```

    {
        // Выводим на консоль, что самолёт задержан
        Console.WriteLine("The plane is delayed");
        // Возвращаем 0, что значит, что не было отправлено ни одного пассажира
        return 0;
    }
}

// Объявлен тип делегата AirlineAction
// Возвращаемое значение типа int, параметров нет
delegate int AirlineAction();

class Program
{
    static void Main(string[] args)
    {
        // Создаём экземпляр Авиакомпании
        Airline myAirline = new Airline();

        // Создаём делегат и назначаем ему метод SendFlight объекта myAirline
        AirlineAction act = myAirline.SendFlight;

        Random rndGen = new Random();
        int rnd = 0;
        do
        {
            // Случайное число
            rnd = rndGen.Next(20);
            // Если не повезло
            if (rnd == 13)
            {
                // Назначаем делегату метод DelayedFlight
                act = myAirline.DelayedFlight;
            }
            // Вызываем делегат, т.е. метод на который он ссылается
            act();
        }
        while (act == myAirline.SendFlight);
        // Цикл продолжается, пока метод делегата - метод SendFlight

    }
}

```

## Сравнение делегатов

Как вы могли заметить, делегат можно сравнивать с методами, а также с другими делегатами такого же типа.

С делегатами другого типа сравнивать нельзя.

```

static void Main(string[] args)
{
    // Авиакомпания №1
    Airline myAirline = new Airline();
    // Авиакомпания №2

```

```

        Airline air2 = new Airline();

        // Первому делегату назначен метод SendFlight объекта myAirline
        AirlineAction act1 = myAirline.SendFlight;

        // Второму делегату назначен метод DelayedFlight объект myAirline
        AirlineAction act2 = myAirline.DelayedFlight;

        // Третьему делегату назначен метод DelayedFlight объект air2
        AirlineAction act3 = air2.DelayedFlight;

        // Сравнивается первый делегат с методом Send Flight объекта myAirline
        if (act1 == myAirline.SendFlight)
        {
            // Именно этот метод был ему назначен, поэтому условие выполнится
            Console.WriteLine("Delegate act1 refers to the method SendFlight of the
object myAirline");
        }
        // Сравнивается первый делегат со вторым делегатом
        if (act1 != act2)
        {
            // Они ссылаются на разные методы, а значит не равны, поэтому условие
выполнится
            Console.WriteLine("Delegates act1 and act2 refers to different
methods");
        }
        // Сравнивается второй делегат с третьим
        // Хотя они оба указывают на метод DelayedFlight
        // Но эти методы принадлежат разным объектам, поэтому считаются разными
        if (act2 != act3)
        {
            // Методы не совпадают, поэтому условие выполнится
            Console.WriteLine("The same methods from different object are not
equal.");
        }
    }
}

```

## Делегат как свойство класса

Добавим в последний пример немного инкапсуляции:

```

// Всё тот же тип делегата
delegate int AirlineAction();

// Всё тот же класс Авиакомпании
class Airline
{
    private static Random gen;

    // Свойство Flight типа делегата AirlineAction (!)
    // Задавать значение которому можно только изнутри этого класса
    public AirlineAction Flight { get; private set; }

    static Airline()
    {
        gen = new Random();
    }
}

```

```

    {
        gen = new Random();
    }

    // Конструктор по умолчанию
    public Airline()
    {
        // Назначает свойству Flight (т.е. делегату)
        // метод SendFlight, который теперь приватный (!)
        Flight = SendFlight;
    }

    // Метод SendFlight стал приватным, т.е. его больше нельзя вызвать
    // напрямую извне
    private int SendFlight()
    {
        int passengers = gen.Next(140, 780);
        Console.WriteLine("The plane leaves with {0} passengers on
board", passengers);
        // Генерируем новое случайное число
        int badLuck = gen.Next(20);
        // Если удача подвела
        if (badLuck == 13)
        {
            // Назначаем свойству Flight (которое имеет тип делегата)
            // Метод DelayedFlight, который также стал приватным
            Flight = DelayedFlight;
        }
        return passengers;
    }

    // Приватный метод DelayedFlight
    private int DelayedFlight()
    {
        Console.WriteLine("The plane is delayed");
        // Возвращает делегату Flight метод SendFlight
        Flight = SendFlight;
        return 0;
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Здесь всё работает так же, как и раньше:
        // Создаётся авиакомпания
        Airline myAirline = new Airline();

        // Переменная для количества пассажиров, которое вернёт Flight
        int passgrs;
        do
        {

```

```

// Вызываем свойство (!) Flight нашей авиакомпании
// Поскольку свойство имеет тип делегата, то его можно вызвать
    passgrs = myAirline.Flight();
// И иногда это будет вызов метода SendFlight, а иногда вызов метода
DelayedFlight
// В зависимости от внутренней логики класса Airline
// Там в принципе может вызываться что-угодно!
    }
while (passgrs != 0);
// Заканчиваем цикл, когда метод в свойстве Flight вернёт 0
    }
}

```

### Делегаты и методы

А теперь самое главное: делегаты можно использовать в качестве аргументов методов, а также в качестве возвращаемых значений.

И это – очень распространенный, гибкий и мощный инструмент.

В аргумент метода типа делегата может быть передан как непосредственно делегат такого типа, так и просто метод подходящего описания.

### Делегат в аргументах метода

```

// Определение типа делегата WorkAction
// С возвращаемым значением int и без параметров
delegate int WorkAction();

// Класс Горожанин
class Citizen
{
    // Приватное поле типа делегата WorkAction
    private WorkAction workAction;
    // Обычное свойство Имя горожанина
    public string Name { get; set; }
    // Статический объект рандома
    public static Random Random { get; set; }

    // Статический конструктор для инициализации генератора случайных чисел
    static Citizen()
    {
        Random = new Random();
    }

    // Конструктор с параметром, для создания горожанина с именем
    public Citizen(string name)
    {
        Name = name;
    }

    // Метод SetWork, который принимает в качестве параметра аргумент типа
    делегата WorkAction
    public void SetWork(WorkAction newWork)
    {

```

```

// Если этот делегат содержит ссылку на метод(ы)
if (newWork != null)
{
    // Записываем его в закрытое поле workAction
    workAction = newWork;
}

// Метод Work
public int Work()
{
    // Создаём переменную, в которой будет сохраняться сумма заработанных
денег
    int earnedMoney = 0;
    // Если рабочего действия нет
    if (workAction == null)
    {
        // Выводим сообщение, что у горожанина нет работы
        Console.WriteLine("The Citizen {0} has no work!", Name);
        return earnedMoney;
    }
    // Случайное число, которое определяет, сколько итераций будет у цикла
работы
    int randomWork = Random.Next(10);
    Console.WriteLine("Beginning of work!");
    // По циклу, столько раз, сколько выпало рандомом
    for (int i = 0; i < randomWork; ++i)
    {
        // Вызывается делегат workAction
        // И значение, которое он возвращает, прибавляется к значению переменной
earnedMoney
        earnedMoney += workAction();
    }
    Console.WriteLine("End of work!");
    Console.WriteLine("Today citizen {0} earned {1} money!", Name,
earnedMoney);
    // Возвращается количество заработанных денег
    return earnedMoney;
}

}

class Program
{
    // Статический метод, описывающий действие по работе писателя
    static int Write()
    {
        Console.WriteLine("Writes something...");
        // Возвращает случайное число - количество денег, полученное за эту
работу
        return Citizen.Random.Next(120, 190);
    }

    // Статический метод, описывающий действие по работе программиста
    static int Programming()

```

```

    {
        Console.WriteLine("Programs back-end...");
        // Также возвращает случайное число
        return Citizen.Random.Next(160, 410);
    }

    static void Main(string[] args)
    {
        // Создаём первого горожанина
        Citizen human1 = new Citizen("Bob");
        // Он пытается работать
        human1.Work();

        // Вызывается метод SetWork, который принимает делегат
        // В качестве параметра мы передаём метод Write,
        // Который по сигнатуре схож с описанием делегата
        // Этот метод Write записывается в закрытое поле класса,
        // Чтобы потом использоваться в методе Work
        human1.SetWork(Write);
        // Отправляем Боба на работу
        human1.Work();

        // Создаём Ричарда
        Citizen human2 = new Citizen("Richard");
        // Пытаемся поработать
        human2.Work();
        // Назначаем действия по работе программиста
        human2.SetWork(Programming);
        // Отправляем на работу 2 раза
        human2.Work();
        human2.Work();

        // Назначаем Бобу тоже работу программиста
        human1.SetWork(Programming);
        // И отправляем его на работу
        human1.Work();
    }
}

```

## Еще пример

```

// Объявляем тип делегата, возвращающий булево значение
// И принимающий два параметра типа int
delegate bool Comparison(int a, int b);

static class ArrayUtils
{
    // Вместо метода для поиска минимального элемента,
    // Метода для поиска максимального элемента, и др.
    // Мы добавили один единственный универсальный метод
    // Условие работы которого определяется вторым аргументом
    // Типа делегата Comparison
    public static int ShowByExpr(int[] arr, Comparison compare)
    {

```



```

if (arr.Length == 0)
{
    Console.WriteLine("Array is empty");
}
int element = arr[0];
for (int i = 1; i < arr.Length; ++i)
{
    // Там, где в алгоритме раньше находилась ключевая проверка
    // Вызывается делегат. И от результата его выполнения зависит вся логика
    // этого метода
    if (compare(arr[i], element))
    {
        element = arr[i];
    }
}
// Возвращаем полученный результат
return element;
}

}

class Program
{
    // Вспомогательный статический метод
    // Для поиска минимального элемента методом ShowByExpr
    static bool Min(int a, int b)
    {
        return a < b;
    }

    // Вспомогательный статический метод
    // Для поиска максимального элемента методом ShowByExpr
    static bool Max(int a, int b)
    {
        return a > b;
    }

    // Вспомогательный статический метод
    // Для поиска минимального положительного элемента методом ShowByExpr
    static bool MinPositive(int a, int b)
    {
        return a > 0 ? Min(a, b) : false;
    }

    // И тут можно написать ещё сколько угодно разных методов,
    // Которые будут как-либо менять поведение метода ShowByExpr

    static void Main(string[] args)
    {
        // Создаётся массив
        int[] array1 = { 10, 17, 23, 6, -5, 14, 0, -11, -2 };

        // Вызывается метод ShowByExpr, в качестве параметра-делегата
        // Передаётся метод Min - благодаря чему метод возвращает минимальное
        // число

```

```

int result = ArrayUtils.ShowByExpr(array1, Min);
    Console.WriteLine("Min value in the array is {0}", result);

// Вызывается метод ShowByExpr, в качестве параметра-делегата
// Передаётся метод Max - благодаря чему метод возвращает максимальное
число
result = ArrayUtils.ShowByExpr(array1, Max);
    Console.WriteLine("Max value in the array is {0}", result);

// Вызывается метод ShowByExpr, в качестве параметра-делегата
// Передаётся метод MinPositive - благодаря чему метод возвращает
минимальное положительное число
result = ArrayUtils.ShowByExpr(array1, MinPositive);
    Console.WriteLine("Min positive value in the array is {0}",
result);
}
}

```

### Анонимные функции

Объявление целых методов в классе довольно громоздко и излишне в том случае, когда эти методы очень короткие и используются лишь для присваивания делегату.

Поэтому в С# существует возможность создавать анонимные функции – безымянные кодовые блоки, служащие для инициализации делегатов.

У анонимных функций имеется две формы записи (старая и новая):

- анонимный метод;
- лямбда-выражение.

### Анонимные методы

Анонимный метод – старый способ создания безымянного блока кода, инициализирующего некоторый конкретный делегат.

Для описания анонимного метода нужно описать обычный функциональный блок после ключевого слова `delegate`:

```

delegate (список_параметров)
{
    // Код метода
}

```

### Пример 1

```

class Program
{
    // Объявлен делегат
    delegate void MessageHandler(string message);

    static void Main(string[] args)
    {
        // Создаётся экземпляр делегата и инициализируется анонимным методом
        MessageHandler handler = delegate (string mes)

```

```

{
// Анонимный метод выводит значение параметра на консоль
    Console.WriteLine(mes);
};
// Делегат вызывается, происходит выполнение анонимного метода
handler("Hello world from anonymous method!");
Console.ReadKey(true);
}
}

```

## Пример 2

```

delegate bool Comparison(int a, int b);

static class ArrayUtils
{
    public static int ShowByExpr(int[] arr, Comparison compare)
    {
        if (arr.Length == 0)
        {
            Console.WriteLine("Array is empty");
        }
        int element = arr[0];
        for (int i = 1; i < arr.Length; ++i)
        {
            if (compare(arr[i], element))
            {
                element = arr[i];
            }
        }
        return element;
    }
}

class Program
{
    // Больше нет ненужных вспомогательных методов

    static void Main(string[] args)
    {
        // Создаётся массив
        int[] array1 = { 10, 17, 23, 6, -5, 14, 0, -11, -2 };

        // Теперь при вызове метода ShowByExpr второй параметр-делегат
        // инициализируется
        // Анонимным методом, описанным прямо здесь
        int result = ArrayUtils.ShowByExpr(array1, delegate (int a, int b) {
            return a < b; });
        Console.WriteLine("Min value in the array is {0}", result);

        // В каждом случае то же самое, по аналогии - анонимный метод
        result = ArrayUtils.ShowByExpr(array1, delegate (int a, int b) { return
        a > b; });
        Console.WriteLine("Max value in the array is {0}", result);
    }
}

```

```
// И здесь свой анонимный метод
    result = ArrayUtils.ShowByExpr(array1, delegate (int a, int b) {
return a > 0 ? a < b : false; });
    Console.WriteLine("Min positive value in the array is {0}",
result);
}
}
```

## Рассмотрим следующий пример

```
// Объявление делегата Sum
delegate int Sum(int number);

class Program
{
// Метод SomeVar, который возвращает делегат Sum
static Sum SomeVar()
{
int result = 0;

// Создаётся экземпляр делегата Sum и ему присваивается
// Анонимный метод
    Sum del = delegate (int number)
    {
// Который считает сумму чисел до заданного
for (int i = 0; i <= number; i++)
    {
        result += i;
    }
// И возвращает её
return result;
    };
// Метод SomeVar возвращает делегат del
return del;
}

static void Main()
{
// Создаётся объект делегата Sum del1 и ему назначается
// Делегат, который вернул метод SomeVar (а он вернул делегат с
анонимным методом)
    Sum del1 = SomeVar();

// Для чисел от 1 до 5 включительно:
for (int i = 1; i <= 5; i++)
    {
// Выводит сумму всех целых положительных чисел до заданного
// Посредством вызова делегата del1
        Console.WriteLine("Сумма {0} равна: {1}", i, del1(i));
    }

    Console.ReadKey(true);
}
}
```

## Замыкания

Вывод на консоль из предыдущего примера может озадачить.

Если присмотреться к коду, можно заметить, что в анонимном методе используется переменная `result`, объявленная во внешнем методе `SomeVar`. Такие переменные называются Захваченными.

А сам механизм захвата окружающих переменных, используемых в анонимном методе, называется Замыканием.



*Рисунок 3.1 – Лямбда-выражения*

## Лямбда-выражения

Лямбда-выражения представляют собой упрощенный синтаксис описания анонимных методов.

Для описания Лямбда-выражения используется лямбда-оператор `=>`.

Существует два вида лямбда-выражений:

- одиночные;
- блочные.

### Одиночные лямбды

Одиночное лямбда-выражение используется, когда действие, выполняемое анонимной функцией, может быть описано одним выражением (действием).

Имеет следующий синтаксис:

параметр => выражение
-----------------------

Или

(список_параметров) => выражение
----------------------------------

Одиночные лямбды по умолчанию возвращают результат описанного в них выражения, поэтому если в анонимной функции предполагался `return` – его писать не нужно.

### Демонстрация

Покажем синтаксис описания лямбда-выражений в сравнении с ранее написанными анонимными методами:

```
// Эквивалентные записи:
// Анонимный метод
MessageHandler handler = delegate (string mes)
{
    Console.WriteLine(mes);
};
// Лямбда
MessageHandler handler = mes => Console.WriteLine(mes);

// Эквивалентные записи:
// Анонимный метод
int result = ArrayUtils.ShowByExpr(array1, delegate (int a, int b) {
    return a < b; });
// Лямбда
int result = ArrayUtils.ShowByExpr(array1, (a, b) => a < b);
```

### Блочные лямбды

Внутри блочных лямбда-выражений может содержаться любое количество выражений, любой код, т.е. нет никаких ограничений. Но за счет этого приходится расплачиваться красотой синтаксиса:

```
параметр => { /* ... */ }

(список_параметров) => { /* ... */return/* ... */ }
```

Тело блочного лямбда-выражения берется в фигурные скобки, и в случае наличия возвращаемого значения в нем необходимо обозначать `return`.

### Пример 1

```
class Program
{
    // Делегат Operation, в качестве параметров принимающий 2 int-a
    delegate int Operation(int x, int y);
    // Делегат Square, в качестве параметра принимающий 1 int
    delegate int Square(int x);

    static void Main(string[] args)
    {
```

```

// Создаём объект делегата и присваиваем ему анонимную функцию
// Заданную лямбда-выражением, принимающую два параметра x и y
// И возвращающую их сумму (x + y)
Operation operation = (x, y) => x + y;

// Выводим на консоль результат вызова делегата operation с параметрами
10 и 20
// Он будет выполнять лямбда-выражение, заданное выше, т.е. посчитает x +
y
Console.WriteLine(operation(10, 20));
// То же самое только для параметров 40 и 20
Console.WriteLine(operation(40, 20));

// Создаём делегат типа Square и назначаем ему лямбда-выражение
// С параметром i, возвращающее квадрат этого числа i
Square square = i => i * i;

// Вызываем делегат square с аргументов 6
// Вызовётся лямбда-выражение, которое возведёт 6 в квадрат
int z = square(6);
// Результат выведется на консоль
Console.WriteLine(z);

Console.ReadKey(true);
}
}

```

## Пример 2

```

// *Пример с кривым и нелогичным дизайном, просто для демонстрации лямбд

// Объявляем 3 типа делегатов, которые понадобятся нам в дальнейшем
delegate int LengthLogin(string s);
delegate bool BoolPassword(string s1, string s2);
delegate void Captcha(string s1, string s2);

// Программа имитирует процесс регистрации пользователя
class Program
{
    // Статический метод Ввода логина
    private static void SetLogin()
    {
        // Пользователя просят ввести логин
        Console.Write("Enter your login: ");
        string login = Console.ReadLine();

        // Создаём объект делегата LengthLogin, которому назначаем лямбда-
выражение
        // Которое принимает строковый параметр и возвращает длину этой строки
        LengthLogin lengthLoginDelegate = s => s.Length;
    }
}

```

```

// Этот делегат вызывается с введённым логином в качестве аргумента
// И возвращает длину этого логина
int lengthLogin = lengthLoginDelegate(login);
// Если логин из более чем 25 символов
if (lengthLogin > 25)
{
    Console.WriteLine("The login is too long.\n");

    // Рекурсия на этот же метод, чтобы попытаться ввести логин заново
    SetLogin();
}

static void Main()
{
    // Вызов метода ввода логина
    SetLogin();

    // Пользователь вводит пароль
    Console.Write("Enter password: ");
    string password1 = Console.ReadLine();
    // Пользователь повторяет пароль
    Console.Write("Please, confirm password: ");
    string password2 = Console.ReadLine();

    // Делегат типа BoolPassword, инициализированный лямбда-выражением
    // Которое принимает две строки и возвращает результат их сравнения на
    равенство
    BoolPassword bp = (s1, s2) => s1 == s2;

    // Если вызов этого делегата вернул true, т.е. два ввода пароля совпадают
    if (bp(password1, password2))
    {
        Random ran = new Random();
        // Имитируем капчу, для проверки на робота
        string resCaptcha = "";
        // Создаём строку из 10 случайных символов
        for (int i = 0; i < 10; i++)
        {
            // Случайное число от 0 до 100, преобразуется к символу с таким кодом
            // И добавляется в конец строки resCaptha
            resCaptcha += (char)ran.Next(0, 100);
        }
        // Пользователь тщетно пытается ввести капчу
        Console.WriteLine("Enter this Captcha to prove that you are not a robot:
" + resCaptcha);
        string resCode = Console.ReadLine();

        // В делегат типа Captcha записывается блочное лямбда-выражение
        // Принимающее 2 параметра строки s1 и s2
        Captcha cp = (s1, s2) =>
        {

```



```

// Если эти строки равны
if (s1 == s2)
{
    // Выводим сообщение об успешной регистрации
    Console.WriteLine("Registration is successfully completed!");
}
else
{
    // Если строки не равны, регистрация не удалась
    Console.WriteLine("We sorry, but it seems that you are a robot");
}

};

// Вызываем делегат ср для проверки капчи
// Передавая параметры эталонной капчи и введённой пользователем попытки
ср(resCaptcha, resCode);

}

else
{
    // Если пароли не совпадают, регистрация не удалась
    Console.WriteLine("Registration failed. Please try again.");
}

    Console.ReadLine(true);
}
}

```

### Пример 3

```

// Определяем обобщённый делегат Predicate
// Возвращающий bool, а принимающий два аргумента типа T (типа-параметра)
delegate bool Predicate<T>(T first, T second);

// Статический класс наших методов для работы с массивом
static class ArrayUtils
{
    // Статический обобщённый метод сортировки массива любого типа
    // Первым параметром принимает массив из элементов типа T,
    // Возвращает также массив из элементов типа T
    // Самое интересное - второй параметр: объект делегата Predicate с типом-
    // параметром T
    // Метод в этом делегате будет использоваться для управления логикой
    // алгоритма сортировки
    public static T[] Sort<T>(T[] array, Predicate<T> predicate)
    {
        // Обычный алгоритм "пузырьковой сортировки"
        for (int i = 0; i < array.Length; ++i)
        {
            // Сравниваем элемент массива "каждый с каждым"
            for (int j = i; j < array.Length; ++j)
            {
                // Используем вызов делегата для i и j элемента массива
                // В условии if-а, который определяет порядок и условие сортировки
            }
        }
    }
}

```

```

    if (predicate(array[i], array[j]))
    {
        // Если предикат вернул true для i-ого и j-ого элемента
        // Они переставляются "стаканчиком"
        T tmp = array[j];
        array[j] = array[i];
        array[i] = tmp;
    }
}

// В конце возвращается отсортированный массив
return array;
}

// Обобщённый статический метод Show для вывода массива любого типа на
консоль
public static void Show<T>(T[] array)
{
    // Объединяет элементы массива в строку при помощи метода Join
    Console.WriteLine(string.Join(", ", array));
}

class Program
{
    static void Main(string[] args)
    {
        // Создаём неупорядоченный массив из целых чисел
        int[] arr1 = { -3, 8, 0, 13, 6, 82, 34, 2, -6, -10, 1, 3 };

        // Вызываем описанный нами метод сортировки Sort для этого массива arr1
        // Во второй параметр типа делегата Predicate передаём лямбда-выражение
        // Которое возвращает результат сравнения чисел a и b (a > b)
        ArrayUtils.Sort(arr1, (a, b) => a > b);
        // С помощью метода Show выводим массив arr1 на консоль
        ArrayUtils.Show(arr1);

        // Создаём неупорядоченный массив чисел с плавающей точкой
        double[] arr2 = { 0.354, -9.234, 0.3, 5.19, -1.628, -7.62, 4.001, 8.971,
        -0.045, 13.58 };

        // Вызываем метод сортировки Sort для этого массива
        // И вторым параметром передаём лямбду, которая возвращает
        // Результат сравнения двух параметров a < b
        ArrayUtils.Sort(arr2, (a, b) => a < b);
        // Выводим то, что получилось
        ArrayUtils.Show(arr2);

        // Третий массив из строк
        string[] arr3 = { "Matthew", "Abraham", "Bill", "Jimmy", "Nancy",
        "Ronald", "Isaac" };
    }
}

```

```

// Вызываем метод сортировки, в качестве делегата передаём лямбда-
выражение
// Которое сравнивает первые символы строк a и b, и возвращает результат
их сравнения a[0] > b[0]
ArrayUtils.Sort(arr3, (a, b) => a[0] > b[0]);
// Выводим отсортированный массив на консоль
    ArrayUtils.Show(arr3);
}
}

```

## Пример 4

```

// Класс Хоккейная команда
public class HockeyTeam
{
    // Приватные поля для Названия и Года основания
    private string _name;
    private int _founded;

    // Свойства для этих полей
    public string Name
    {
        get { return _name; }
        private set { _name = value; }
    }

    public int Founded
    {
        get { return _founded; }
        private set { _founded = value; }
    }

    // Конструктор
    public HockeyTeam(string name, int year)
    {
        Name = name;
        Founded = year;
    }
}

public class Example
{
    public static void Main()
    {
        Random rnd = new Random();
        // Создаём список хоккейных команд
        List < HockeyTeam > teams = new List< HockeyTeam > ();
        // Добавляем в него 6 объектов команд
        teams.AddRange(new HockeyTeam[] { new HockeyTeam("Detroit Red
Wings", 1926),

```

```

new HockeyTeam("Chicago Blackhawks", 1926),
new HockeyTeam("San Jose Sharks", 1991),
new HockeyTeam("Montreal Canadiens", 1909),
new HockeyTeam("St. Louis Blues", 1967) });
// Массив годов для поиска
int[] years = { 1920, 1930, 1980, 2000 };
// Выбираем из этого массива случайный год
int foundedBeforeYear = years[rnd.Next(0, years.Length)];
Console.WriteLine("Teams founded before {0}:",
foundedBeforeYear);
// Ищем в списке команды, основанные до выбранного года
// Встроенный метод списка FindAll принимает в качестве параметра делегат,
// Анонимную функцию, которая будет вызываться для каждого элемента из
этого списка
// Иначе говоря, FindAll вернёт список из тех элементов списка, для которых
условие
// В переданном делегате выполняется (возвращает true)
foreach (var team in teams.FindAll(x => x.Founded <= foundedBeforeYear))
{
    // В цикле перебираем все элементы полученного списка, и выводим
информацию о командах
    Console.WriteLine("{0}: {1}", team.Name, team.Founded);
}
}
}

```

## Встроенные делегаты

Вообще-то, в .NET есть набор встроенных обобщенных типов делегатов, которые можно применять практически в любой ситуации, когда нужны делегаты.

Т.е. нет необходимости объявлять свои типы делегатов.

Встроенные типы делегатов:

- Action;
- Predicate;
- Func.

### Делегаты Action

Встроенный делегат Action принимает до 16 параметров любых типов и не имеет возвращаемого значения.

Что значит до 16 параметров? Для этого делегата описано 17 перегрузок для любого количества параметров до 16.

Описаны делегаты Action следующим образом:

```

publicdelegate void Action();
publicdelegate void Action<inT1>(T1 obj1);
publicdelegate void Action<inT1, inT2>(T1 obj1, T2 obj2);
// Ит.д. до
publicdelegate void Action<inT1, inT2, inT3, inT4, inT5, inT6, inT7, inT8,

```

```
inT9, inT10, inT11, inT12, inT13, inT14, inT15, inT16>(T1 obj1, T2 obj2,
    T3 obj3, T4 obj4, T5 obj5, T6 obj6, T7 obj7, T8 obj8, T9 obj9, T10
obj10, T11 obj11,
    T12 obj12, T13 obj13, T14 obj14, T15 obj15, T16 obj16);
```

## Использование Action

```
// Статический метод Operation, принимающий два числа и делегат типа
Action<int, int>
// Т.е. делегат с двумя параметрами типа int и без возвращаемого значения
staticvoid Operation(int x1, int x2, Action<int, int> op)
{
    // Если первое число больше второго
    if (x1 > x2)
    {
        // Вызываем делегат из третьего параметра, передавая ему эти 2 числа x1 и
x2
        op(x1, x2);
    }
}

// Использование встроенного делегата Action вместо описания собственного
делегата:
// delegate void MyDelegate(int a, int b);

staticvoid Main(string[] args)
{
    // Создаём объект делегата Action<int, int>
    Action<int, int> op;
    // Назначаем ему лямбда-выражение, выводящее на консоль сумму параметров
    op = (x, y) => Console.WriteLine("Суммачисел: " + (x + y));
    // Вызываем статический метод Operation, передавая 2 числа и делегат op
    Operation(10, 6, op);
    // Переназначаем делегату новое лямбда-выражение для разности чисел
    op = (x, y) => Console.WriteLine("Разность чисел: " + (x1 - x2));
    // Вызываем метод Operation, передаём делегат op
    Operation(10, 6, op);

    Console.ReadKey(true);
}
```

## Делегат Predicate

Обобщенный делегат Predicate существует лишь в одной вариации, и принимает параметр любого типа, а возвращает bool.

Как правило, он используется для различных методов фильтрации коллекций, например, поиска элементов по заданному предикату.

Описан как:

```
publicdelegateboolPredicate<inT>(T obj);
```

## Предикаты в методах списка

```
class Program
{
    static void Main(string[] args)
    {
        // Создаём список из целых чисел, сразу инициализируем его
        var list = new List<int>() { 5, 2, 3, 34, 12, 10, 5, 9, 2 };
        // Выводим список на консоль
        Console.WriteLine(string.Join(", ", list));
        // Метод списка Exists принимает в качестве параметра делегат типа Predicate
        // Этот метод проверяет, есть ли в списке элемент, удовлетворяющий заданному предикату выражению
        // В качестве делегата мы передаём лямбда-выражение x => x == 10
        // Которое возвращает результат сравнения числа с десяткой
        // Значит, этот метод будет последовательно выполнять это выражение (сравнивать с 10)
        // Каждый элемент списка. Если встретится элемент, для которого сравнение выполнится,
        // Метод вернёт true, если не встретится - вернёт false
        if (list.Exists(x => x == 10))
        {
            // Если метод вернул true, выводим сообщение на консоль
            Console.WriteLine("Элемент со значением 10 присутствует в списке");
        }

        // Метод списка FindAll возвращает новый список из тех элементов старого списка,
        // Которые удовлетворяют заданному делегату (предикату)
        var valuesGrater10 = list.FindAll(x => x > 10);
        Console.WriteLine("Список из элементов, меньших чем 10:");
        // Выводим список на консоль
        Console.WriteLine(string.Join(", ", valuesGrater10));

        // Метод списка RemoveAll удаляет из списка все элементы, удовлетворяющие переданному делегату
        // Здесь ему передаётся лямбда-выражение x => x % 2 == 0
        // Те элементы списка, для которых это лямбда-выражение вернёт true
        // Будут удалены из списка
        list.RemoveAll(x => x % 2 == 0);
        Console.WriteLine("Список после удаления всех чётных элементов:");
        // Выводим список на консоль
        Console.WriteLine(string.Join(", ", list));

        // Метод списка TrueForAll возвращает true, если для всех элементов этого списка
        // Переданный делегат (предикат) вернёт true
        // Т.е. в данном случае проверяется, все ли элементы списка больше нуля
    }
}
```

```

bool allElementsPositive = list.TrueForAll(x => x > 0);
// Если это true
if (allElementsPositive)
{
    // То выводим соответствующее сообщение на консоль
    Console.WriteLine("В списке только положительные значения");
}
}
}

```

## Делегат Func

Делегаты Func похожи на делегаты Action. Отличие заключается в наличии у Func возвращаемого значения, также задающегося параметрически.

Описания делегатов Func:

```

publicdelegate TResult Func<out TResult>();
publicdelegate TResult Action<in T, out TResult>(T arg);
publicdelegate TResult Action<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
publicdelegate TResult Action<in T1, in T2, in T3, out TResult>(T1 arg1, T2
arg2, T3 arg3);
// И т.д.

```

## Пример

```

// Класс Постройка
class Building
{
    // Свойство Тип
    public string Type { get; set; }

    // Конструктор
    public Building(string type)
    {
        Type = type;
    }
}

// Класс Строитель
class Builder
{
    // Метод "Строить". Возвращает true или false, в зависимости от того,
    // Было ли завершено строительство в срок
    // Два параметра: первый - объект постройки, которую нужно Строить
    // Второй - делегат, принимающий параметр типа Постройки и возвращающий
    // целое число -
    // Количество дней, за которое строитель построит данную конструкцию
    public bool Build(Building building, Func<Building, int> process)
    {
        // Выполняем метод переданный в делегате
        // Помещаем значение, которое он вернул, в переменную timeSpend
        int timeSpend = process(building);
    }
}

```

```

// Если это значение меньше или равно 90
if (timeSpend <= 90)
{
    // Всё нормально
    Console.WriteLine("Building complete.");
    return true;
}
else
{
    // Иначе - строитель не уложился в срок
    Console.WriteLine("All deadlines broken... This is a bad builder.");
    return false;
}
}

}

class Program
{
    static void Main(string[] args)
    {
        // Создаётся два объекта постройки: больница и аэропорт
        Building building1 = new Building("Hospital");
        Building building2 = new Building("Airport");

        Random rnd = new Random();

        // Объекту делегата присваиваем блочное лямбда-выражение
        // (аргумент - Building, возвращаемое значение - int)
        // Это функция для нормального процесса выполнения строительства
        Func<Building, int> normalWork = b =>
        {
            // Объявляем переменную для количества дней работ
            int days;
            // Если тип строения - больница
            if (b.Type == "Hospital")
            // Выбираем количество дней случайным образом от 50 до 80
                days = rnd.Next(50, 80);
            // Иначе если аэропорт
            elseif (b.Type == "Airport")
            // Выбираем количество дней от 50 до 400
                days = rnd.Next(50, 400);
            else
            // Строить что-то другое наши строители не умеют
                days = 100500;
            // Возвращаем количество дней
            return days;
        };
        // Ещё один делегат такого же типа
        // Только с другим лямбда-выражением
        // Которое описывает процесс ленивой некачественной работы
        Func<Building, int> lazyWork = b =>
        {
            int days;

```



```

// Если тип постройки - больница
if (b.Type == "Hospital")
// Срок строительства растягивается до максимально допустимого
количество дней
    days = 90;
else
// Что-либо другое строить не хотим
    days = 100500;
// Возвращаем количество дней
return days;
};

// Создаём четырёх строителей
    Builder mr1 = new Builder();
    Builder mr2 = new Builder();
    Builder mr3 = new Builder();
Builder mr4 = new Builder();

// Первый строитель строит Больницу работая нормально
// Первый параметр постройка building1, тип которой - Больница
// Второй параметр - делегат для нормальной работы normalWork
    mr1.Build(building1, normalWork);
// Второй строитель строит Аэропорт, работая нормально
    mr2.Build(building2, normalWork);
// Третий строитель нехотя строит Больницу
    mr3.Build(building1, lazyWork);
// Четвёртый строитель ленится строить Аэропорт
mr4.Build(building2, lazyWork);
    }
}

```

## Применение делегатов

Делегаты обычно применяются:

- для фильтрации и обработки коллекций,
- в качестве Callback (обратного вызова),
- в событиях.

Первый пункт мы уже видели, а вот с оставшимися двумя еще не сталкивались.

### Callback

Callback (или Обратный вызов) – механизм передачи одной функции в качестве параметра другой функции, когда переданная функция вызывается в определенный момент (обычно, в конце) работы вызываемой функции.

Если упростить, то Callback – это когда функция передается в другую функцию, и выполняется в конце ее работы.

По сути, ничего нового.

## Область применения

Callback обширно распространены в языках программирования на основе цикла событий (JS), при асинхронном и многопоточном программировании и программировании распределенных систем.

Например, когда имеется какая-либо продолжительная операция, допустим, печать документа на принтере, после которой необходимо выполнить некоторый код, этот код передается в качестве обратного вызова в метод печати.

```
Callback = { действия_которые_нужно_выполнить_после_печати };
Печать(документы, Callback);
// Callback будет выполнен, когда закончится печать документов
```

Интересный пример на тему использования Callback можно найти в Приложении Ж.

## Паттерн Decorator

```
class Program
{
    // У нас есть метод, который принимает другой метод (Action func)
    // И цвет консоли ConsoleColor color
    // Этот метод будет оборачивать другой метод в метод-обёртку
    public static Action WrapWithColor(Action func, ConsoleColor color)
    {
        // Внутри этого метода описан локальный метод
        // Это метод-обёртка, в которую обернут переданный в качестве параметра
        func метод
        void Wrap()
        {
            // Этот метод создаёт замыкание и захватывает значения переменных func и
            color
            var oldColor = Console.ForegroundColor;
            // Меняет цвет консоли на color
            Console.ForegroundColor = color;
            // Вызывает метод func
            func();
            // Возвращает старый цвет консоли
            Console.ForegroundColor = oldColor;
        }
        // Внешний метод возвращает локальный метод как возвращаемое значение
        return Wrap;
    }

    // Просто метод с выводом на консоль
    public static void PrintMessage()
    {
        Console.WriteLine("This is message");
        Console.WriteLine("On 2 lines.");
    }

    public static void Main(string[] args)
```

```

{
    // Мы вызываем метод WrapWithColor и передаём в него метод PrintMessage,
    а также синий цвет
    // И этот метод возвращает нам другой метод - обёртку, в которой
    находится старый метод PrintMessage
    var printGreenMessage = WrapWithColor(PrintMessage, ConsoleColor.Blue);

    // То же самое, только с жёлтым цветом
    var printYellowMessage = WrapWithColor(PrintMessage,
    ConsoleColor.Yellow);

    // С голубым цветом
    var printCyanMessage = WrapWithColor(PrintMessage, ConsoleColor.Cyan);

    // И вызываем полученные методы
    // Каждый метод выведет сообщение своим цветом
    printGreenMessage();
    printYellowMessage();
    printCyanMessage();
}
}

```

## События

События (event) – механизм для задания реакции на какие-либо действия, происходящие в классе.

По своей сути, события в C# представляют собой просто делегат в виде свойства класса.

Тогда функции, которые назначаются этому делегату, называются Обработчиками события, и описывают то, что будет происходить при наступлении такого события.

С использованием механизма события часто выстраивают взаимодействие различных систем между собой, события повсеместно применяются в разработке графических интерфейсов.

Вообще, события – очень распространенный инструмент в объектно-ориентированном программировании, и их можно увидеть на каждом шагу.

## Описание события в классе

Хотя событие по сути является просто свойством типа делегата, чтобы разграничить обычные свойства и события, для последних был введен свой особенный синтаксис.

Событие в классе описывается через ключевое слово event с методами аксессуарами add и remove.

По аналогии с простым свойством и аксессуарами get и set.

```

// Полная форма записи
private event WorkAction reachedEndStation;
public event Action ReachedEndStation

```

```

{
    add { reachedEndStation += value; }
    remove { reachedEndStation -= value; }
}

// Сокращённая форма записи
public event Action ReachedEndStation;

```

### Вызов события

Чтобы событие стало событием, оно должно где-то происходить.

Значит, где-то в коде класса, в котором событие описано, в том месте, где логически происходит ожидаемое действие, с которым ассоциировано это событие, должен быть описан его вызов:

```

if (this.ReachedEndStation != null)
{
    this.ReachedEndStation();
}

```

### Обработчик события

Обработчик события (функция) назначается событию, как обычному делегату.

Это, как правило, происходит уже за пределами класса, в котором событие описано.

```

// Описание метода-обработчика
public void OnSubwayReachedEndStation1()
{
    Console.WriteLine("All right, run the next!");
}

// ...

// Назначение обработчика событию
metro.ReachedEndStation += OnSubwayReachedEndStation1;

```

### Делегат для события и тип аргументов

Выше в качестве делегата для события мы использовали Action.

Но обычно для событий в C# создаются специальные делегаты следующего формата:

```

delegate void Какой_нибудь_EventHandler(object sender,
    Какие_нибудь_EventArgs);

```

Тип возвращаемого значения – всегда void.

Первый параметр типа object – для источника события.

Второй параметр собственного типа, производного от EventArgs, для дополнительных параметров события.

Пример программы с использованием событий можно найти в приложении 3.

### **Событие консоли CancelKeyPress**

Во многих стандартных классах есть свои события. Например, в классе Console имеется событие CancelKeyPress.

Это событие нажатия клавиши завершения программы, и происходит оно в тот момент, когда пользователь нажимает сочетание клавиш Ctrl+C.

```
publicstaticvoid Main(string[] args)
{
    Console.CancelKeyPress += Console_CancelKeyPress;
    while (true) ;
}

privatestaticvoid Console_CancelKeyPress(object sender,
ConsoleCancelEventArgs e)
{
    Console.WriteLine("Cancel KeyPress");
}
```

## Приложение А

### Пример реализации обобщенного класса Словаря

```
// Обобщённый класс Словаря с двумя типами-параметрами Т и U
// Реализован на основе циклического списка
// Причём на тип Т наложено ограничение по реализации интерфейса
// Это нужно для того чтобы можно было корректно сравнивать элементы
этого типа
public class Dict<T, U>
where T : IEquatable<T>
{
    // Приватный класс Элемент словаря
    private class DictNode
    {
        // Указатель на следующий элемент
        public DictNode Next { get; set; }
        // Указатель на предыдущий элемент
        public DictNode Prev { get; set; }
        // Ключ, по которому осуществляется доступ
        // Имеет тип параметр Т
        public T Key { get; set; }
        // Значение типа U
        public U Data { get; set; }
    }

    // Ссылка на какой-нибудь элемент для доступа
    private DictNode head;

    // Свойство для количества элементов
    public int Count { get; set; }

    // Конструктор по умолчанию
    public Dict()
    {
        head = null;
        Count = 0;
    }

    // Приватный метод поиска элемента DictNode по ключу
    // Ключ имеет тип параметр Т
    private DictNode FindElementByKey(T key)
    {
        // Перебираем все элементы
        for (int i = 0; i < Count; ++i)
        {
            // Двигаем указатель
            head = head.Prev;
            // Если находим совпадение ключей
            if (head.Key.Equals(key))
            {
                // Возвращаем найденный элемент
                return head;
            }
        }
    }
}
```

```

    }
    // Если ничего не нашли, возвращаем null
    return null;
}

// Приватный метод Добавления или замены значения по ключу
private void AddOrReplace(T key, U value)
{
    // Пытаемся найти элемент с таким ключом
    DictNode found = FindElementByKey(key);
    // Если не нашли
    if (found == null)
    {
        // Создаём новый элемент
        DictNode newElement = new DictNode();
        newElement.Data = value;
        newElement.Key = key;
        // Если наш словарь пуст
        if (head == null)
        {
            // Записываем новый элемент в указатель списка
            head = newElement;
            // Изацикливаем его
            head.Next = head.Prev = head;
        }
        else
        {
            // Если в словаре уже что-то есть
            // Добавляем новый элемент перед head
            head.Prev.Next = newElement;
            newElement.Prev = head.Prev;
            newElement.Next = head;
            head.Prev = newElement;
        }
        // Увеличиваем количество на 1
        ++Count;
    }
    else
    {
        // Когда элемент с заданным ключом найден в словаре
        // Просто перезаписываем у него значение на новое
        found.Data = value;
    }
}

// Публичный индексатор для доступа к элементам словаря по ключу
// Возвращает значение, т.е. тип U, принимает индекс-ключ, т.е. тип T
public U this[T key]
{
    get
    {
        // Пытаемся найти элемент с заданным ключом
        DictNode found = FindElementByKey(key);
        // Если не находим
    }
}

```

```

        if (found == null)
        {
            // Возвращаем значение по умолчанию для типа значения U
            returndefault(U);
        }
        // А если находим, возвращаем из него значение
        return found.Data;
    }
    set
    {
        // Добавляем или заменяем значение по ключу
        AddOrReplace(key, value);
    }
}

// Публичный метод Вывода на консоль
publicvoidPrint()
{
    Console.WriteLine("Dictionary:");
    // Перебираем все элементы словаря
    for (int i = 0; i < Count; ++i)
    {
        // Перемещаем указатель
        head = head.Next;
        // И выводим каждый элемент на консоль
        Console.WriteLine(" [{0}] : {1}", head.Key, head.Data);
    }
    Console.WriteLine();
}

}

classProgram
{
    staticvoid Main(string[] args)
    {
        // Создаём новый объект класса Dict с типом ключа string и типом значения
        float
        Dict<string, float> dict = new Dict<string, float>();
        // Добавляем в него элементы через индексатор
        dict["pi"] = 3.1415926f;
        dict["e"] = 2.7182818284f;
        dict["Na"] = 6.022e+23f;
        dict["h"] = 6.6260e-34f;
        dict["G"] = 6.674e-11f;
        dict["c"] = 299_792_458f;

        // Выводим словарь на консоль
        dict.Print();
    }
}

```



## Приложение Б

### Тестовая программа «Метро» с применением Callback

```
// Класс поезда метро
class Metro
{
    // Константный словарь с перечислением всех станций текущей линии метро
    // Ключ - число, номер станции. Значение - строка, название станции
    private readonly Dictionary<int, string> Stations = new Dictionary<int,
string>()
    {
        { 1, "Dodger Stadium" },
        { 2, "Children's Hospital & Hollywood Presbyterian Medical
Center" },
        { 3, "Kaiser Permanenter Hospital" },
        { 4, "The Grove" },
        { 5, "Farmers Market" },
        { 6, "Park La Brea" },
        { 7, "LA Country Museum of Art" },
        { 8, "Petersen Automotive Museum" },
        { 9, "Washington/Fairfax Transit Hub" },
        { 10, "Beverly Center" },
        { 11, "ULCA" }
    };

    // Свойство - номер станции, на которой сейчас находится поезд
    // Сеттер приватный
    public int CurrentStation { get; private set; }

    // Конструктор, задаёт начальную станцию
    public Metro()
    {
        CurrentStation = 1;
    }

    // *Интересный метод
    // Принимает 2 параметра: объект перечисления цвета консоли ConsoleColor
    // И делегат Action
    // Этот метод выступает в роли обёртки для вызова других методов
    private void PrintWithColor(ConsoleColor color, Action action)
    {
        // Запоминаем старый цвет консоли
        var oldColor = Console.ForegroundColor;
        // Меняем цвет консоли на новый, переданный в виде аргумента
        Console.ForegroundColor = color;
        // И вызываем принятый в делегате метод
        if (action != null)
            action();
        // А после его выполнения меняем цвет консоли на прежний
        Console.ForegroundColor = oldColor;
    }
}
```

```

    // Т.е. весь вывод, который будет происходить в переданном в делегат
    // методе будет цвета color
    }

    // Метод Запуска поезда метро
    // Первый параметр speed - скорость поезда
    // Второй и третий параметры - коллбэки, типа делегата Action<Metro>
    // Делегат onSuccess будет вызываться, когда поезд метро успешно доедет
    // до конечной станции
    // А делегат onAccident - когда произойдёт авария во время движения
    // Для коллбэков заданы значения по умолчанию None - т.е. они
    // необязательные параметры
    public void Run(int speed, Action<Metro> onSuccess = null, Action<Metro>
onAccident = null)
    {
        // Если скорость не положительная
        if (speed <= 0)
        {
            // У поезда будут проблемы с передвижением
            Console.WriteLine("And how the train would run with speed {0}?.",
speed);
            return;
        }

        // Специальная формула, описывающая вероятность аварии поезда в
        // зависимости от его скорости
        // Максимальная скорость - 120, авария 100%
        // С уменьшением скорости вероятность аварии уменьшается по этой сложной
        // формуле
        int crashProbability = speed <= 120 ? Convert.ToInt32(4 * Math.Sqrt(120 -
speed) + 350 / Math.Sqrt(speed) - 31) : 1;

        Random generator = new Random();

        // Делегат типа Func, принимающий один параметр типа int и возвращающий
        // объект типа int
        // Поезд может двигаться по линии в 2 направлениях
        // И вместо того, чтобы писать 2 условия в самом цикле движения, или, ещё
        // хуже, написать 2 разных цикла
        // Здесь характер движения (порядок смены станций) описывается вот этим
        // делегатом
        // По умолчанию назначаем делегату лямбда-выражение, которое увеличивает
        // на 1 переданный аргумент (номер станции)
        // Эта логика подходит, когда нужно двигаться от 1 станции к 11
        Func<int, int> getNextStationByDirection = x => ++x;
        // Если же поезд сейчас находится на последней 11 станции
        if (CurrentStation == Stations.Count)
        {
            // Назначаем делегату лямбда-выражение с противоположным действием -
            // уменьшением номера станции
            getNextStationByDirection = x => --x;
        }

        // 2 переменные, которые будут использоваться внутри цикла движения
        int danger = 0, nextStation = 0;

```

```

    // Рассчитываем время (для метода Sleep), которое будет занимать движение
    поезда между станциями
    // Чем меньше скорость speed, тем дольше будет происходить поездка
    (длиннее Sleep)
    int timeInRun = (1800 - speed * speed / 8);
    // Проверяю, чтобы время было адекватным
    if (timeInRun < 0)
    {
        timeInRun = 0;
    }
    // Цикл езды
    do
    {
        // Получаем номер следующей станции, через ранее определённый делегат
        getNextStationByDirection
        // Который будет уменьшать или увеличивать номер, в зависимости от
        направления движения
        nextStation = getNextStationByDirection(CurrentStation);
        // Выводим строчку на консоль, с какой станции на какую сейчас едет поезд
        Console.WriteLine("The subway train started moving from the \"{0}\"
        station to the \"{1}\" station.\n",
                           Stations[CurrentStation],
                           Stations[nextStation]);
        // Получаем случайное число от 0 до значения crashProbability
        danger = generator.Next(0, crashProbability);
        // Спим
        Thread.Sleep(timeInRun);
        // Выводим шум езды

        Console.WriteLine("Wwhhhhhooooooooooooooooooooooooooooommm...\n");
        // Спим
        Thread.Sleep(timeInRun);
        // Если случайное число выпало равным 0 - значит произошла авария
        if (danger == 0)
        {
            // Используем метод-обёртку PrintWithColor, для отображения на консоли
            определённых строк
            // Красным цветом.
            // Цвет ConsoleColor.Red передаётся первым параметром
            // Вывод, который будет отображаться красным, передаём вторым параметром
            в виде блочной лямбды
            PrintWithColor(ConsoleColor.Red, () =>
            {
                // Поезд попадает в аварию
                Console.WriteLine("CCCC-R-R-A-ACCKK-K-K!!!");
                Console.WriteLine("Oh no! The train flew off the
                rail!!\n");
            });
            // По аналогии, вызываем метод PrintWithColor, только теперь уже для
            зелёного цвета
            PrintWithColor(ConsoleColor.Green, () =>
            {
                // В качестве делегата теперь передаётся этот код:

```

```
// Если в делегате есть метод
if (onAccident != null)
// Вызов метода-коллбэка onAccident
// Т.е. вот только что в коде произошла авария поезда
// И поэтому этот делегат вызывается именно в этом месте, сразу после аварии
// В качестве параметра типа Metro передается текущий объект this
    onAccident(this);
});
// Раз произошла авария, выходим из метода, не продолжая движения
return;
}
// Если аварии не было, продолжаем езду

Console.WriteLine("Ushhhuuuuuuuuuuuuuuuuuuuuuuuuuuuhhhh...\n");
Thread.Sleep(timeInRun);
// Меняю текущую станцию на следующую
CurrentStation = nextStation;
// Бипаем
    Console.Beep();
// Пишем, что приехали на эту станцию
Console.WriteLine("The train arrived at the station \"{0}\".\n\n",
Stations[CurrentStation]);
Thread.Sleep(1000);
// И продолжаем ехать дальше по циклу
}
while (CurrentStation != 1 && CurrentStation != Stations.Count);
// Пока текущая станция не станет первой или последней на линии
// Тогда выведем, что это конечная станция
Console.WriteLine("End station.");
// И вызовем делегат-коллбек onSuccess
if (onSuccess != null)
// Т.к. момент настал - поезд успешно доехал
onSuccess(this);
}
}

class SubwayOrganisation
{
public static void Main(string[] args)
{
// Создаём объект поезда метро
Metro metro = new Metro();
// Попытка проехать со скоростью 150. Коллбэки не используются
metro.Run(150);

// Описываем делегат типа Action<Metro>
// Который передадим в качестве коллбэка методу Run
// Собственно метод для вызова спасателей
Action<Metro> rescueCall = (m) =>
{
// Выводит всякие сообщения
Console.WriteLine("People need help!");
```

```

        Console.WriteLine("Rescuers are on their way...");
        Thread.Sleep(3000);
        // В общем, спасатели всех спасают
        Console.WriteLine("They arrived and rescued the wounded people!");
        Console.WriteLine("Now everything is fine.");
    };
    // Вызываем метод Run, и теперь используем коллбэки
    // Первый параметр - скорость, 100
    // Второй параметр, коллбэк для успешного завершения езды
    // Т.е. то, что будет выполнено, когда, и если, поезд успешно доедет до
конечной
    // Туда мы передаём такое лямбда-выражение: (m) => m.Run(100, onAccident:
rescueCall)
    // Которое заново рекурсивно вызывает метод Run, с такой же скоростью, и
с одним
    // Коллбэком на случай аварии. Т.е. просто чтобы по достижению конечной
станции на линии
    // Поезд начинал движения обратно на начальную станцию
    // В качестве коллбэка для ситуации с аварией в оба вызова Run передаётся
    // Описанный выше делегат rescueCall, для вызова спасателей
    metro.Run(100, (m) => m.Run(100, onAccident: rescueCall), rescueCall);
}
}

```

## Приложение В

### Тестовая программа «Метро» с использованием событий

```
// Класс Аргументов для события Аварии метро
// Производный от класса EventArgs
// Имена классов аргументов принято заканчивать словами EventArgs
publicclassMetroAccidentEventArgs : EventArgs
{
    // Свойство для описания участка, на котором произошла авария
    // Тип: кортеж из двух интов - номера станций, между которыми была авария
    public (int, int) AccidentArea { get; set; }

    // Свойство Количество пострадавших
    publicint NumberOfWounded { get; set; }

    // Конструктор
    public MetroAccidentEventArgs((int, int) area, int wounded)
    {
        AccidentArea = area;
        NumberOfWounded = wounded;
    }
}

// Класс для аргументов события Достижения конечной станции
publicclassMetroEndStationEventArgs : EventArgs
{
    // Свойство Станция (может быть первой, или последней)
    publicint Station { get; set; }

    // Свойство Скорость, с которой ехал поезд
    publicint OldSpeed { get; set; }

    // Конструктор
    public MetroEndStationEventArgs(int station, int speed)
    {
        Station = station;
        OldSpeed = speed;
    }
}

// Делегат для события Авария
// Имя принято заканчивать словами EventHandler
// Первый параметр object sender - источник события
// Второй параметр - для передачи сопутствующих параметров этого события
publicdelegatevoidMetroAccidentEventHandler(object sender,
MetroAccidentEventArgs e);

// Делегат для события Достижение конечной станции
publicdelegatevoidMetroEndStationEventHandler(object sender,
MetroEndStationEventArgs e);
```

```

// Класс поезда метро
class Metro
{
    // Сделано публичным и статическим, чтобы можно было получать доступ
    // извне без экземпляра класса
    public static readonly Dictionary<int, string> Stations = new
    Dictionary<int, string>()
    {
        { 1, "Dodger Stadium" },
        { 2, "Children's Hospital & Hollywood Presbyterian Medical
Center" },
        { 3, "Kaiser Permanenter Hospital" },
        { 4, "The Grove" },
        { 5, "Farmers Market" },
        { 6, "Park La Brea" },
        { 7, "LA Country Museum of Art" },
        { 8, "Petersen Automotive Museum" },
        { 9, "Washington/Fairfax Transit Hub" },
        { 10, "Beverly Center" },
        { 11, "ULCA" }
    };

    public int CurrentStation { get; private set; }

    // В класс добавлено событие ReachedEndStation - Достижение конечной
    // станции
    // Имеет тип созданного для него выше делегата
    MetroEndStationEventHandler
    public event MetroEndStationEventHandler ReachedEndStation;

    // Событие Accident - Авария
    public event MetroAccidentEventHandler Accident;

    public Metro()
    {
        CurrentStation = 1;
    }

    private void PrintWithColor(ConsoleColor color, Action action)
    {
        var oldColor = Console.ForegroundColor;
        Console.ForegroundColor = color;
        if (action != null)
            action();
        Console.ForegroundColor = oldColor;
    }

    // Убраны параметры метода коллбэки
    public void Run(int speed)
    {
        if (speed <= 0)
        {

```

```

        Console.WriteLine("And how the train would run with speed
{0}?.", speed);
    return;
}
    int crashProbability = speed <= 120 ? Convert.ToInt32(4 * Math.Sqrt(120 -
speed) + 350 / Math.Sqrt(speed) - 31) : 1;
    if (crashProbability < 1)
    {
        crashProbability = 1;
    }

    Random generator = new Random();

    Func<int, int> getNextStationByDirection = x => ++x;
    if (CurrentStation == Stations.Count)
    {
        getNextStationByDirection = x => --x;
    }

    int danger = 0, nextStation = 0;
    int timeInRun = (1800 - speed * speed / 8);
    if (timeInRun < 0)
    {
        timeInRun = 0;
    }
    do
    {
        nextStation = getNextStationByDirection(CurrentStation);
        Console.WriteLine("The subway train started moving from the
\"{0}\" station to the \"{1}\" station.\n",
            Stations[CurrentStation],
            Stations[nextStation]);
        danger = generator.Next(0, crashProbability);
        Thread.Sleep(timeInRun);
        Console.WriteLine("Wwhhhhhoooooooooooooooooooooommm... \n");
        Thread.Sleep(timeInRun);
        if (danger == 0)
        {
            PrintWithColor(ConsoleColor.Red, () =>
            {
                Console.WriteLine("CCCR-R-R-A-ACCKK-K-K!!!");
                Console.WriteLine("Oh no! The train flew off the
rail!!\n");
            });
            // Добивилась переменная woundedPeople, обозначающая количество
            пострадавших в аварии людей
            int woundedPeople = generator.Next(40, 500);
            PrintWithColor(ConsoleColor.Green, () =>
            {
                // Теперь, вместо проверки и вызова коллбэка
                // У нас проверяется и вызывается событие
                if (Accident != null)
                {

```





```

        Metro.Stations[e.AccidentArea.Item1],
Metro.Stations[e.AccidentArea.Item2]);
        Console.WriteLine("People need help!");
        Console.WriteLine("Rescuers are on their way...");
        Thread.Sleep(3000);
        // А вот тут используется второе свойство аргументов этого события,
        количество пострадавших NumberOfWounded
        Console.WriteLine("They arrived and rescued {0} wounded people!",
e.NumberOfWounded);
        Console.WriteLine("Now everything is fine.");
    }

    // Обработчик для второго события, ReachedEndStation
    publicstaticvoid OnEndStation(object sender, MetroEndStationEventArgs e)
    {
        // Поскольку источник события принят в виде типа object,
        // Чтобы работать с ним, как с Metro, нужно выполнить преобразование
        if (sender is Metro metro)
        {
            // Вызывается метод Run, объекта-источника события
            // И в него передаётся значение из аргументов этого события OldSpeed
            // Т.е. поезд метро отправляется в движение в обратном направлении с той
            же скоростью
            metro.Run(e.OldSpeed);
        }
    }

    publicstaticvoid Main(string[] args)
    {
        Metro metro = new Metro();
        metro.Run(150);

        // Назначение обработчика OnAccident событию Accident
        metro.Accident += OnAccident;
        // Назначение обработчика OnEndStation событию ReachedEndStation
        metro.ReachedEndStation += OnEndStation;
        metro.Run(100);
    }
}

```

## ИСПОЛЬЗУЕМАЯ ЛИТЕРАТУРА

1. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Пер. с англ. – 6-е изд. – М. : Вильямс, 2013. – 1312 с.
2. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#/ Пер. с англ. – 4-е изд., исправ. – М. : Издательство «Русская редакция»; СПб. : Питер, 2019. – 898 с.
3. OOP Principles, <https://www.slideshare.net/ssuser8f1bf3/oop-principles-37419002> – [Электронный ресурс], Дата доступа: 17.10.2018.
4. Язык программирования C# и .NET, <https://metanit.com/sharp/general.php> – [Электронный ресурс], Дата доступа: 03.02.2019.
5. C# 5.0 и платформа .NET 4.5, [https://professorweb.ru/my/csharp/charp\\_theory/level1/infocsharp.php](https://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php), [Электронный ресурс], Дата доступа: 24.02.2019

Учебное издание

**Карпович Павел Сергеевич**

## **С# в примерах**

Практикум в двух частях.

### **Часть II**

Практикум для слушателей  
повышения квалификации и переподготовки

Редактор *Д.С. Ковалевский*  
Редактор технический *Ю.Э. Недбальская*  
Компьютерная верстка *Ю.Э. Недбальская*  
Корректор *Д.С. Ковалевский*

Подписано в печать 16.12.2019. Формат 60×84 1/16. Бумага офсетная  
Ризография. Усл. печ. л. 6,05. Уч.-изд. л. 3,45.  
Тираж 50 экз. Заказ 668.

Выпущено по заказу ГУО «Республиканский институт  
повышения квалификации и переподготовки работников  
Министерства труда и социальной защиты Республики Беларусь».

Издатель и полиграфическое исполнение:  
учреждение образования  
«Минский государственный ПТК полиграфии».

Свидетельство о государственной регистрации издателя,  
изготовителя и распространителя печатных изданий  
№ 1/129 от 27.12.2013.

Ул. В. Хоружей, 7, 220005, г. Минск.