Урок №5. Структуры, перечисления, делегаты, события

1.Структуры

1.1. Понятие структуры

В C#.NET структуры предназначены для группирования и хранения небольших порций данных.

Структуры – это типы значений, а не ссылочные типы. Это значит, что они либо сохраняются в стеке, либо являются встроенными (последнее – если они являются частью другого объекта, хранимого в куче), и имеют те же ограничения времени жизни, что и простые типы данных.

1.2. Синтаксис объявления структуры

Для объявления структуры используют ключевое слово `struct`:

```
struct Имя : Интерфейсы
{
    // объявления членов
}
```

Приведем пример работы со структурами. Напишем программу, которая будет работать с геометрическими фигурами, вычислять их периметр и площадь.

```
struct Dimensions
{
    public double Length;
    public double Width;
}
```

В структурах, как и в классах, возможно определение полей, свойств, методов, конструкторов, итераторов и событий. Например:

```
struct Dimensions
{
    public double Length;
    public double Width;
    public Dimensions(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Diagonal
    {
        get
        {
            return Math.Sqrt(Length * Length + Width * Width);
        }
    }
}
```

Для передачи методу ссылки на структуру используются ключевые слова `ref` и `out`.

`ref` – предоставляет возможность изменить значения полей структуры внутри другого метода, до вызова метода структура должна быть инициализирована:

```
class RefExample
{
    public void DoubleDimensions(ref Dimensions d)
    {
        d.Length *= 2;
        d.Width *= 2;
    }
    public void DoSomething()
    {
        Dimensions d = new Dimensions(2, 3);
        DoubleDimensions(ref d);

        // теперь d.Length равно 4, а d.Width равно 6;

        DoSomethingElse();
    }
}
```

`out` – используется при необходимости получения из метода значения одного или нескольких параметров, которые, до вызова метода, не были инициализированы:

```
class OutReturnExample
{
    public void OutReturnExampleMethod(out int i, out string s1,
        out string s2)
    {
        i = 44;
        s1 = "I've been returned";
        s2 = null;
    }
    public void DoSomething()
    {
        int value;
        string str1, str2;
        OutReturnExampleMethod(out value, out str1, out str2);
        // value теперь равняется 44
         // str1 равно "I've been returned"
         // str2 осталось равным null;

        DoSomethingElse();
    }
}
```

1.3. Необходимость и особенности применения структур

Структуры используют для оптимизации производительности, когда из функциональности класса нужно только хранение некоторых небольших порций данных.

Тип `struct` не требует отдельной ссылочной переменной. Это означает, что при использовании структур расходуется меньший объем памяти, и благодаря прямому доступу

к структурам, при работе с ними не снижается производительность, что имеет место при доступе к объектам классов.

При объявлении все поля структуры автоматически инициализируются нулевыми значениями.

За счет хранения в стеке выделение и удаление памяти под структуры происходит очень быстро.

При присваивании одной структуры другой или при передаче методу структуры, как параметра, происходит полное копирование содержимого структуры, что намного медленнее передачи ссылки. (Скорость копирования зависит от размера структуры: чем больше полей – тем дольше копирование).

Структуры – это типы значений и наследуются от `System.ValueType`.

Структуры можно трактовать как упрощенные классы.

Структуры имеют доступ к методам класса `System.Object`, которые могут переопределять. Структуры не поддерживают наследование. Структуры могут наследовать интерфейсы. Компилятор всегда генерирует конструктор по умолчанию без параметров, который переопределить невозможно.

Выделение памяти под всю структуру происходит при объявлении переменной. Соответственно, операция `new` для структур действует иначе, чем для ссылочных типов: она вызовет инициализацию полей значениями по умолчанию.

```
public void DoSomething()
{
    Dimensions d;
    // память выделена, но поля значениями не инициализированы;
    d = new Dimensions();
    // d.Length равно 0, d.Width равно 0;

    d.Length += 2;
    d.Length += 4;

    // d.Length равно 2, а d.Width равно 4;
}
```

Конструктор по умолчанию инициализирует все поля нулевыми значениями. Также невозможно обойти конструктор по умолчанию, определяя начальные значения полей. Следующий код вызовет ошибку компиляции:

```
struct Dimensions
{
    public double Length = 3;
    public double Width = 4;
}
```

Структуру можно снабдить `Close()` или `Dispose()` – так же, как это делается с классами.

2. Перечисления (enum)

2.1. Понятие перечисления.

Перечисление (enumeration) – это непустой список именованных констант. Он задает все значения, которые может принимать переменная данного типа. Перечисления являются классом и наследуются от базового класса `System.Enum`.

2.2. Синтаксис объявления перечисления.

Для объявления перечисления используется ключевое слово `enum`, указывается имя перечисления и в фигурных скобках перечисляются имена констант:

```
enum EnumName {elem1, elem2, elem3, elem4}
```

Примером может быть перечисление дней недели:

```
enum DayOfWeek
{
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}
```

Следующий пример перечисления описывает возможные типы грузовых автомобилей:

```
enum TransportType
{
    Semitrailer, Coupling, Refrigerator, OpenSideTruck, Tank
}
```

Обращение к элементу перечисления осуществляется указыванием имени класса перечисления и через точку имени конкретного элемента перечисления.

```
DayOfWeek day = DayOfWeek.Monday;
```

Константы перечисления имеют целочисленный тип. По умолчанию, первой константе присваивается значение 0, а значение каждой следующей константы увеличивается на единицу. В приведенном примере перечисления `DayOfWeek` значения констант будут следующие: `Monday` = 0, `Tuesday` = 1, `Wednesday` = 2, `Thursday` = 3, `Friday` = 4, `Saturday` = 5, `Sunday` = 6. С переменными типов перечислений можно осуществлять арифметические операции. Приведем пример метода возвращающего следующий день недели:

```
public DayOfWeek NextDay(DayOfWeek day)
{
    return (day < DayOfWeek.Sunday) ? day++ : DayOfWeek.Monday;
}
```

Также можно присвоить значение константе явно. Например, следующее перечисление описывает размер скидки для разных типов клиентов:

```
enum Discount
{
    Default = 0, Incentive = 2, Patron = 5, VIP = 15
}
```

2.3. Необходимость и особенности применения перечисления

Перечисления позволяют сделать процесс программирования более удобным и быстрым, помогают избавиться от путаницы при присвоении переменным значений. Можно выделить следующие полезные функции перечислений:

- перечисления гарантируют, что переменным будут присваиваться только разрешенные, ожидаемые значения. Если вы попробуете присвоить экземпляру перечисления значение, которое не входит в список допустимых значений, то компилятор выдаст ошибку;

- перечисления делают код более понятным, так как мы обращаемся не просто к числам, а к осмысленным именам;

- перечисления позволяют программисту сэкономить время. Когда вы захотите присвоить экземпляру перечисляемого типа какое-то значения, то интегрированная в Visual Studio среда IntelliSense отобразит список всех допустимых значений;

- как уже упоминалось, перечисления наследуются от базового класса `System.Enum`, что позволяет вызывать для них ряд полезных методов (более подробно это будет рассмотрено в пункте 2.5).

В основном, перечисления используются, когда нужно передавать соответствующее значение методу и проходить по возможным значениям с помощью оператора `switch`.

Рассмотрим пример распределения товаров, которые необходимо доставить службой доставки в пункт назначения. Предприятие-перевозчик принимает товар, определяет его тип, и, в зависимости от типа товара, определяет каким транспортом необходимо его перевозить. Код структуры описывающей единицу груза может иметь следующий вид:

```
struct ConsignmentItem
{
    public String Name;             // Наименование
    public Decimal Weigth;          // Вес
    public Decimal Price;           // Заявленная стоимость
    public Dimensions Dimensions;   // Размеры
    public CommodityType Type;      // Тип товара
}
```

Структура `Dimensions` рассмотрена выше в разделе 1. Структуры, а перечисление `CommodityType` описывается кодом:

```
enum CommodityType
{
    FrozenFood, Food, DomesticChemistry, BuildingMaterials
}
```

Для описания транспорта создадим структуру `Transport`:

```
struct Transport
{
    public String Name;                          // Название
    public Decimal Capacity;                     // Вместимость
    public TransportType Type;                   // Тип
    public List<ConsignmentItem> Commodities;    // Перечень товаров
}
```

Перечислением `TransportType` опишем типы грузовиков:

```
enum TransportType
{
    Semitrailer, Coupling, Refrigerator, OpenSideTruck, Tank
}
```

Тогда код метода выбора нужного грузовика для товара и добавления в список грузов будет следующим:

```
class EnumSwitchExample
{
    Transport Refrigerator = new Transport();
    Transport Semitrailer = new Transport();
    Transport Coupling = new Transport();
    Transport OpenSideTruck = new Transport();

    public void SetTransport (ConsignmentItem item)
    {
        switch (item.Type)
        {
            case CommodityType.FrozenFood:
                Refrigerator.Commodities.Add(item);
                break;
            case CommodityType.Food:
                Semitrailer.Commodities.Add(item);
                break;
            case CommodityType.DomesticChemistry:
                Coupling.Commodities.Add(item);
                break;
            case CommodityType.BuildingMaterials:
                OpenSideTruck.Commodities.Add(item);
                break;
            default:
                Semitrailer.Commodities.Add(item);
                break;
        }
    }
}
```

2.4. Установка базового типа перечисления

Под базовым типом понимается тип констант перечисления. Как уже упоминалось, по умолчанию перечисления основываются на типе `int`. Но можно создать перечисление на

основе любого из целочисленных типов: byte, sbyte, short, ushort, int, uint, long, ulong. Для этого при объявлении перечисления после его имени указывается через двоеточие нужный тип:

```
enum NameEnum [:базовыйТип] {EnumList}
```

Для примера приведем перечисление, описывающее среднюю дистанцию планет солнечной системы от солнца в километрах:

```
enum DistanceSun : ulong
{
    Sun = 0,
    Mercury = 57900000,
    Venus = 108200000,
    Earth = 149600000,
    Mars = 227900000,
    Jupiter = 7783000000,
    Saturn = 1427000000,
    Uranus = 2870000000,
    Neptune = 4496000000,
    Pluto = 5946000000
}
```

2.5. Использование методов для перечислений

Как уже упоминалось, перечисления являются классом, и наследуется от базового класса System.Enum. Это значит, что для них можно использовать методы данного класса, то есть методы сравнения значений перечисления, методы преобразования значений перечисления в строковое представление, методы преобразования строкового представления значения в перечисление и другие.

Методы	Описание
`int CompareTo(object target)`	Сравнивает текущий экземпляр с заданным объектом `target` и возвращает сведения об их относительных значениях. Возвращает: - значение меньше 0, если значение текущего экземпляра меньше значения `target`; - `0`, если значения равны; - значение больше 0, если значение текущего экземпляра больше значения `target`.
`bool Equals(object obj)`	Переопределен. Возвращает значение, показывающее, равен ли текущий экземпляр заданному объекту `obj`. Возвращает `true`, если равны, иначе – `false`.
`string Format(Type enumType, object value, string format)`	Статический. Преобразует указанное значение заданного перечисляемого типа в эквивалентное строчное представление в соответствии с заданным форматом
`void Finalize()`	Позволяет данному объекту попытаться освободить ресурсы и выполнить другие завер-

	шающие операции, перед тем как объект будет уничтожен в процессе сборки мусора.
`int GetHashCode()`	Возвращает хэш-код для данного экземпляра.
`string GetName(Type enumType, Object value)`	Выводит имя константы в указанном перечислении `enumType`, имеющем заданное значение `value`.
`string[] GetNames(Type enumType)`	Статический. Выводит массив имен констант в указанном перечислении enumType.
`Type GetType()`	Возвращает тип текущего экземпляра.
`TypeCode GetTypeCode()`	Возвращает базовый тип `TypeCode` для этого экземпляра
`Type GetUnderlyingType(Type enumType)`	Статический. Возвращает базовый тип указанного перечисления `enumType`.
`Array GetValues(Type enumType)`	Статический. Выводит массив значений констант в указанном перечислении `enumType`.
`Boolean IsDefined(Type enumType, Object value)`	Статический. Возвращает признак наличия константы с указанным значением `value` в заданном перечислении `enumType`. Возвращает `true`, если константа присутствует, иначе – `false`.
`Object MemberwiseClone()`	Создает копию вызывающего объекта, в которую копируются все члены класса, но при этом не копируются объекты, на которые ссылаются эти члены.
`Object Parse(Type enumType, String value)` `Object Parse(Type enumType, String value, Boolean ignoreCase)`	Статический. Перегружен. Преобразует строковое представление имени или числового значения одной или нескольких перечисляемых констант в эквивалентный перечисляемый объект.

`Object ToObject(Type type, Byte value)`
`Object ToObject(Type type, Int16 value)`
`Object ToObject(Type type, Int32 value)`
`Object ToObject(Type type, Int64 value)`
`Object ToObject(Type type, Object value)`
`Object ToObject(Type type, SByte value)`
`Object ToObject(Type type, UInt16 value)`
`Object ToObject(Type type, UInt32 value)`
`Object ToObject(Type type, UInt64 value)`
	Статический. Перегружен. Возвращает экземпляр указанного типа перечисления, равный заданному значению.

`String ToString()`
`String ToString (IFormatProvider provider)`
`String ToString(String format)`
`String ToString(String format, IFormatProvider provider)`
	Перегружен. Преобразует значение этого экземпляра в эквивалентное ему строковое представление.

3. Делегаты

3.1. Понятие делегата

Делегат – это объект, который может ссылаться на метод. То есть делегат может хранить ссылку на метод и с ее помощью вызывать данный метод. Технически делегат – это ссылочный тип, инкапсулирующий метод с указанной сигнатурой и возвращаемым типом.

3.2. Синтаксис объявления делегата

Для объявления делегата используют ключевое слово `delegate`. После него указывается возвращаемый тип, имя делегата и в скобках сигнатура делегируемых методов:

```
delegate Type DelegateName(list)
```

`Type` - возвращаемый тип, `NameDelegate` – имя делегата, `list` – список параметров, необходимых методам при вызове их с помощью делегата (этот список может быть пустым, то есть делегат может вызывать метод, который не имеет аргументов).

Объявленный делегат может вызвать только те методы, которые возвращают указанный тип `Type` и принимает список параметров `list`. Никаких других методов делегат вызвать не может. Делегат может вызывать как методы экземпляров класса, так и статические методы. Также к делегатам можно применять один из модификаторов доступа: `public`, `protected`, `private`, `internal`, `protected internal`.

Рассмотрим следующий пример. Пускай есть структура `Person`:

```
private struct Person
{
    public String FirstName;
    public String LastName;
    public DateTime BirthDay;

    public Person(String firstName, String lastName,
        DateTime birthDay)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.BirthDay = birthDay;
    }

    public override String ToString()
    {
        return String.Format(
            "Имя: {0}; Фамилия: {1}; Дата рождения: {2:d}.",
            FirstName, LastName, BirthDay);
    }

    public static String GetTypeName() { return "Человек"; }
}
```

Объявим делегат:

```
private delegate string GetAsString();
```

Создадим объект делегата `GetAsString getStringMethod`. Последующие присвоения делегату методов `person.ToString()` и `Person.GetTypeName()` показывает, что при вызове выполняется присвоенный ему метод.

```
static void Main(string[] args)
{
    DateTime birthDay = new DateTime(1978, 2, 15);
    Person person = new Person("Иван", "Петров", birthDay);

    GetAsString getStringMethod =
        new GetAsString(birthDay.ToLongDateString);
    Console.WriteLine(getStringMethod());

    getStringMethod = person.ToString;
    Console.WriteLine(getStringMethod());

    getStringMethod = Person.GetTypeName;
    Console.WriteLine(getStringMethod());
}
```

В результате выполнения в консоль будут выведены строки:

```
15 февраля 1978 г.
Имя: Иван; Фамилия: Петров; Дата рождения: 15.02.1978.
Человек
```

3.3. Цели и задачи делегатов

Бывают ситуации, когда методу нужно передать в качестве параметров не данные, а некоторый метод, или во время компиляции неизвестно, какой именно метод нужно вызвать для обработки данных. Здесь не обойтись без делегатов. Например, это может понадобиться в следующих случаях:

– в C# имеется возможность запускать параллельно на выполнение несколько последовательностей операций. Одна такая последовательность называется потоком. Запуск потока производится с помощью метода `Start()`. Ему необходимо передать метод, с которого должен начинаться новый поток (об этом более подробно будет рассказано в следующих уроках);

– при работе с графическим интерфейсом пользователя используют события. Мы заранее не знаем, какие действия произведет пользователь и соответственно, какие методы вызвать. Поэтому необходимо методу, который обрабатывает событие, передать в качестве параметра делегат.

Главным достоинством делегатов является то, что позволяют определить, какой метод нужно вызвать не на этапе компиляции, а на этапе выполнения. Это полезно, когда необ-

ходимо создать базовую конструкцию (программу), а потом в нее добавлять компоненты. Также делегаты обеспечивают правильность сигнатуры вызываемого метода. Еще одним важным свойством делегата является способность поддерживать события (подробно будет рассмотрено в разделе 4).

Рассмотрим пример реализации сортировки массива хорошо известным «Пузырьковым» методом:

Сортировка осуществляется статическим методом `Sort` класса `BubbleSorter`:

```
static class BubbleSorter
{
    static public void Sort(Object[] array, Comparer comparer)
    {
        for (Int32 i = 0; i < array.Length; i++)
            for (Int32 j = i + 1; j < array.Length; j++)
                if (comparer(array[j], array[i]))
                {
                    Object buffer = array[i];
                    array[i] = array[j];
                    array[j] = buffer;
                }
    }
}
```

Данный метод принимает массив объектов и делегат `Comparer`, который сравнивает два элемента массива:

```
public delegate Boolean Comparer(Object elem1, Object elem2);
```

Для реализации примера создадим массив структур `Person`

```
public struct Person
{
    public String FirstName;
    public String LastName;
    public DateTime Birthday;

    public Person(String firstName, String lastName,
        DateTime birthday)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.Birthday = birthday;
    }

    public override String ToString()
    {
        return String.Format(
            "Имя: {0,-10} Фамилия: {1,-10} Дата рождения: {2:d}.",
            FirstName, LastName, Birthday);
    }
}
```

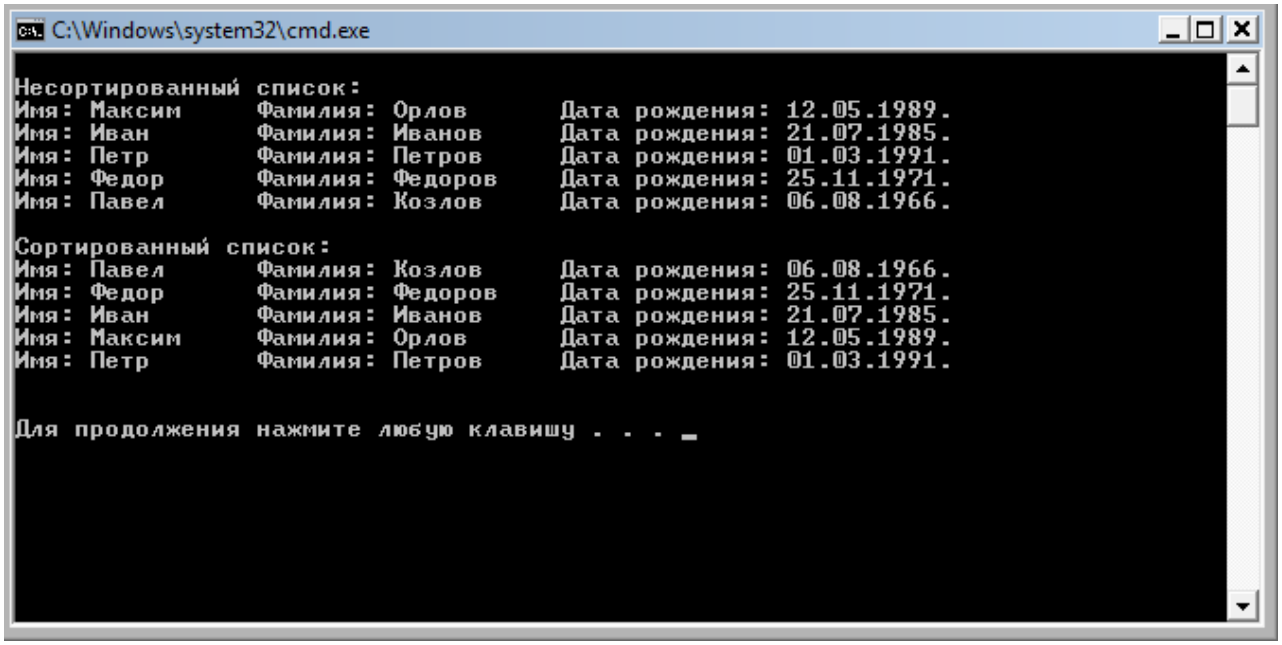
И реализуем метод сравнения двух людей по дате рождения:

```
static public Boolean PersonBirthdayComparer(Object person1, Object person2)
{
    return ((Person)person1).Birthday < ((Person)person2).Birthday;
}
```

Тогда главный метод будет иметь следующий вид:

```
static void Main(string[] args)
{
    Person p0 = new Person("Максим", "Орлов", new DateTime(1989, 5, 12));
    Person p1 = new Person("Иван", "Иванов", new DateTime(1985, 7, 21));
    Person p2 = new Person("Петр", "Петров", new DateTime(1991, 3, 1));
    Person p3 = new Person("Федор", "Федоров", new DateTime(1971, 11, 25));
    Person p4 = new Person("Павел", "Козлов", new DateTime(1966, 8, 6));

    Object[] persons = { p0, p1, p2, p3, p4};

    Console.WriteLine("\nНесортированный список:");
    foreach (Object person in persons) Console.WriteLine(person);

    Console.WriteLine("\nСортированный список:");
    BubbleSorter.Sort(persons, new Comparer(PersonBirthdayComparer));
    foreach (Object person in persons) Console.WriteLine(person);

    Console.WriteLine("\n");
}
```

Результат выполнения программы следующий:



3.4. Вызов нескольких методов через делегат (multicasting)

Иногда возникает необходимость вызвать из одного делегата два или более методов. Не стоит путать с ситуацией, когда создается несколько делегатов, каждый из которых вызывает один метод.

Делегаты в C# поддерживают множественное делегирование (multicasting), то есть возможность вызвать из одного делегата несколько методов. Для реализации множественного делегирования делегат должен возвращать тип `void`. Добавление методов к списку вызовов делегата осуществляется с помощью операции `+=`.

Аналогично, с помощью операции -= метод можно удалить из цепочки делегата. Также можно объединять делегаты с помощью операции +. Тогда получим новый делегат, который будет вызывать методы объединенных делегатов.

Для примера рассмотрим класс `Circle`, в котором описаны четыре метода выводящих в консоль радиус, диаметр, длину окружности и площадь круга:

```
private class Circle
{
    public Double Radius;

    public Circle(Double radius) { this.Radius = radius; }

    public void PrintRadius()
    {
        Console.WriteLine("Радиус равен :           {0,7:N3}", Radius);
    }

    public void PrintDiameter()
    {
        Double diameter = 2 * Radius;
        Console.WriteLine("Диаметр равен :          {0,7:N3}", diameter);
    }

    public void PrintLenght()
    {
        Double lenght = 2 * Math.PI * Radius;
        Console.WriteLine("Длина окружности равна : {0,7:N3}", lenght);
    }

    public void PrintSquare()
    {
        Double square = Math.PI * Radius * Radius;
        Console.WriteLine("Площадь круга равна :    {0,7:N3}", square);
    }
}
```

Создадим делегат, не принимающий параметров и возвращающий тип `void`:

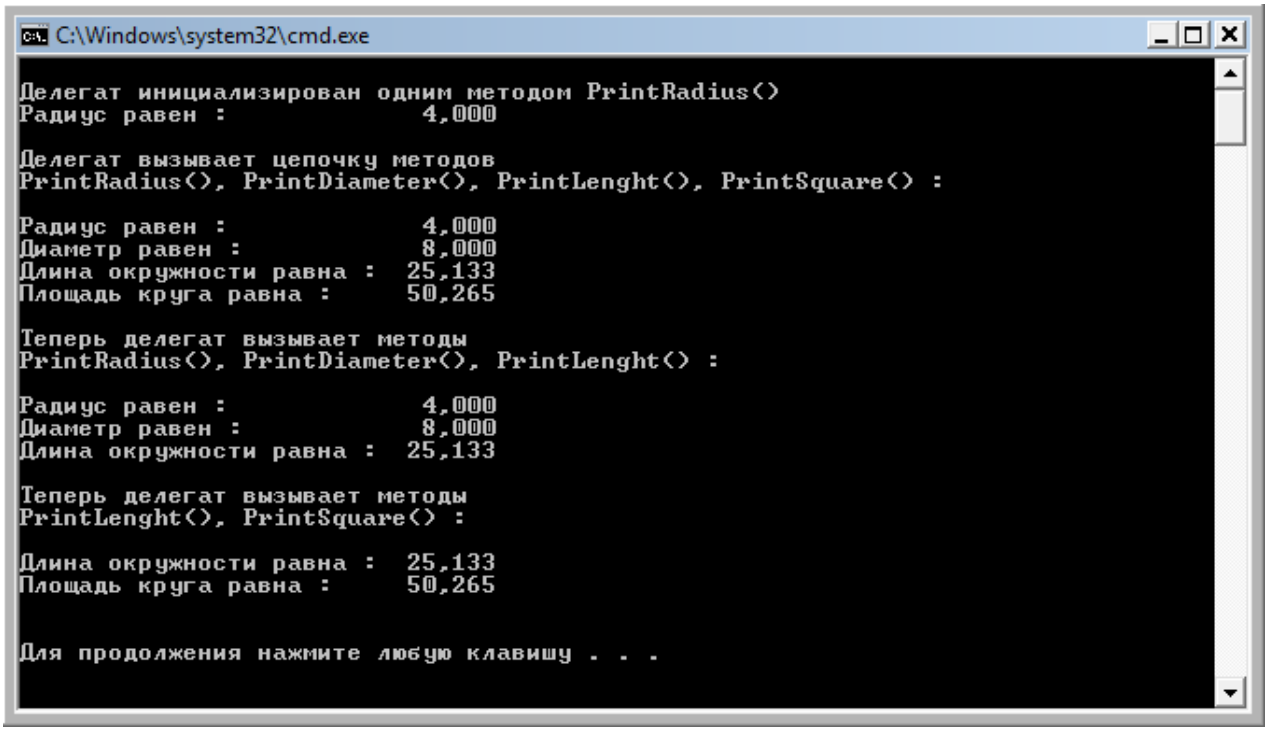
```
private delegate void PrintInfo();
```

Следующий код демонстрирует возможность делегата добавлять и удалять методы из списка вызовов:

```
static void Main(string[] args)
{
    Circle circle = new Circle(4);

    PrintInfo printInfo = new PrintInfo(circle.PrintRadius);
    Console.WriteLine("\nДелегат инициализирован одним методом"+
        " PrintRadius()");
    printInfo();

    printInfo += circle.PrintDiameter;
    printInfo += circle.PrintLenght;
    printInfo += circle.PrintSquare;
```

```
    Console.WriteLine("\nДелегат вызывает цепочку методов \nPrintRadius(),"+
        " PrintDiameter(), PrintLenght(), PrintSquare() :\n");
    printInfo();

    PrintInfo printInfo1 = circle.PrintSquare;

    printInfo -= printInfo1;
    Console.WriteLine(
        "\nТеперь делегат вызывает методы \nPrintRadius(),"+
        " PrintDiameter(), PrintLenght() :\n");
    printInfo();


    printInfo = printInfo + printInfo1 - new PrintInfo(circle.PrintRadius)
        - new PrintInfo(circle.PrintDiameter);
    Console.WriteLine("\nТеперь делегат вызывает методы" +
        " \nPrintLenght(), PrintSquare() :\n");
    printInfo();
}
```

Результат выполнения программы следующий:



Важно помнить, что порядок выполнения методов прикрепленных к одному делегату является неопределенным, то есть заранее нельзя однозначно сказать, в каком порядке они будут выполняться. Поэтому, если необходимо, чтобы методы выполнялись в строго определенном порядке, то multicasting лучше не использовать.

3.5. Базовые классы для делегатов

System.Delegate

Класс `Delegate` является базовым для типов делегатов. Однако только система и компиляторы могут явно наследовать классы `Delegate` и `MulticastDelegate`. Также недопустимо создавать новые типы, производные от типа делегата. Класс `Delegate` не рассматривается как тип делегата, а является классом, наследуемым создаваемыми типами делегатов.

Класс `Delegate` имеет следующие методы:

Методы, свойства и операторы	Описание
`Object Clone()`	Создает неполную копию вызывающего делегата
`Combine`	Перегружен. Сцепляет списки вызовов заданных групповых (комбинируемых) делегатов.
`Delegate CombineImpl(Delegate d)`	Сцепляет списки вызовов заданного группового (комбинируемого) делегата d и текущего группового (комбинируемого) делегата.
`Delegate CreateDelegate`	Перегружен. Создает делегат указанного типа.
`Object DynamicInvoke(params Object[] args)`	Динамически вызывает метод, представленный текущим делегатом. `args` - массив объектов, которые передаются в качестве аргументов методу, представленному текущим делегатом. Если метод, представленный текущим делегатом, не требует аргументов, то вместо `args` передается `null`.
`bool Equals(Object obj )`	Определяет возвращают ли текущий делегат и `obj` одинаковые типы и используют одни и те же методы. Возвращает `true`, если это так, иначе – `false`.
`int GetHashCode()`	Возвращает хэш-код текущего делегата.
`Delegate[] GetInvocationList()`	Основываясь на внутреннем списке ссылок на функции, строится соответствующий массив описателей типов функций. Попытка применения метода к пустому делегату приводит к возникновению исключения
`MethodInfo GetMethodImpl()`	Возвращает статический метод, который представлен текучим делегатом.
`Type GetType()`	Возвращает тип текущего объекта.
`Object MemberwiseClone()`	Создает неполную копию текущего объекта.
`Delegate Remove(Delegate source, Delegate value)`	Удаляет элементы внутреннего списка ссылок на функции.
`string ToString()`	Возвращает строку, которая описывает вызывающий объект.
`bool operator ==( Delegate d1, Delegate d2)`	Возвращает `true`, если делегаты равны, иначе – `false`.
`bool operator !=(Delegate d1,`	Возвращает `true`, если делегаты неравны, ина-

`Delegate d2)`	че – `false`.
`MethodInfo Method { get; }`	Это свойство возвращает метод, на который указывает текущий делегат.
`Object Target { get; }`	Возвращает имя класса, если делегат указывает на нестатический метод. Возвращает значение `null`, если делегат указывает на статический метод.

System.MulticastDelegate

Любой класс-делегат наследует System.MulticastDelegate. Это обстоятельство определяет многоадресность делегатов: в ходе выполнения приложения объект-делегат способен запоминать ссылки на произвольное количество функций независимо от их статичности, нестатичности или принадлежности классам. Многоадресность обеспечивается внутренним списком, в котором хранятся ссылки на функции соответствующие заданной сигнатуре и спецификации возвращаемого значения.

4.События

4.1. Понятие события

Использование событий – это один из наиболее встречающихся в .NET приемов ООП. Можно сказать, что событие – это извещение о возникновении некоторого действия. Это может быть нажатие кнопки, выбор элемента списка и т.д.

Отправителем события может быть любой компонент формы, либо само приложение. Отправитель – это объект, который генерирует событие. Так объект-отправитель оповещает остальные объекты о том, что произошло определенное действие. Получить уведомление о событии может любой другой объект, которому это необходимо. Будем называть такие объекты получателем события. Для того чтобы объект знал о возникновении события, его необходимо подписать на это событие. Это означает, что нужно задать метод-обработчик этого события. Причем сигнатура этой функции должна полностью совпадать с сигнатурой обрабатываемого события. Важно отметить, что отправитель сообщения не знает о существовании получателя, не имеет на него ссылок.

Для реализации событий используются делегаты. Объект-отправитель определяет делегат, а каждый объект-получатель добавляет свой метод-обработчик в цепочку ссылок делегата. Вызов цепочки методов осуществляет объект-отправитель, инициируя выполнение всех методов. Порядок вызова методов подписанных объектов однозначно не определен.

4.2. Синтаксис объявления события

Для объявления события используют ключевое слово `event`:

```
event DelegateName EventName;
```

`DelegateName` – имя используемого делегата; `EventName` – имя события.

Напомним, что объект-отправитель определяет делегат, который должен быть реализован объектами-подписчиками. При возникновении события, методы подписчиков (обработчики события) вызываются через этот делегат. Если на событие нет подписанных объектов, то оно имеет значение `null`, а его вызов вызовет ошибку выполнения.

Существует соглашение, по которому делегаты обработчиков событий возвращают тип `void` и имеют два параметра. Первый – объект-отправитель, то есть объект генерирующий событие (тип `object`), а второй – объект содержащий информацию и параметры события, производный от класса `EventArgs`.

Для примера рассмотрим класс `Person`, который имеет метод `Work()` и событие, не имеющее параметров, `WorkEnded` информирующее о завершение выполнения работы:

```
public class Person
{
    public event EventHandler WorkEnded;

    public String FirstName;
    public String LastName;
    public DateTime Birthday;

    public Person(String firstName, String lastName, DateTime birthday)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.Birthday = birthday;
    }

    public void Work()
    {
        // Do something

        if (WorkEnded != null) WorkEnded(this, EventArgs.Empty);
    }
}
```

Для создания события был использован стандартный делегат `EventHandler` использующийся для генерации событий без параметров. В методе `Main()` показано создание объекта класса `Person` и подписывание класса `Program` на событие `WorkEnded`:

```
static void Main(string[] args)
{
    Person person = new Person("Bill", "Gordon", new DateTime(1962, 8, 11));
    person.WorkEnded += new EventHandler(Person_WorkEnded);

    person.Work();
}
```

Обработчик события имеет следующий код:

```
static void Person_WorkEnded(object sender, EventArgs e)
{
    Console.WriteLine("Работа выполнена ! \n");
}
```

4.3. Необходимость и особенности применения событий

Главное достоинство событий в том, что они позволяют сделать объект-отправитель и объекты-получатели независимыми друг от друга, так как они отделены делегатом. Это делает код более гибким, позволяет разработчику менять реакцию на событие, добавлять новых подписчиков.

4.4. Применение события для многоадресатного делегата

Как уже упоминалось, делегаты могут быть многоадресными. Аналогично, одно событие может активизировать несколько обработчиков, даже если они определены в других объектах. Рассмотрим следующий пример. Создадим класс `Ticker`, его метод `RunTiker()` генерирует событие `TickEvent` каждые 10 мс:

```
public class Ticker
{
    public event TickEventHandler TickEvent;

    private Boolean isEnabled = false;
    public Boolean IsEnabled
    {
        get { return isEnabled; }
        set { isEnabled = value; }
    }

    private Int32 ticks = 0;

    public void RunTiker()
    {
        while (isEnabled && TickEvent != null)
        {
            ticks++;
            TickEvent(this, new TickerArgs(ticks));
            Thread.Sleep(10);
        }
    }
}
```

Событие основано на делегате `TickEventHandler`:

```
public delegate void TickEventHandler(Object sender, TickerArgs args);
```

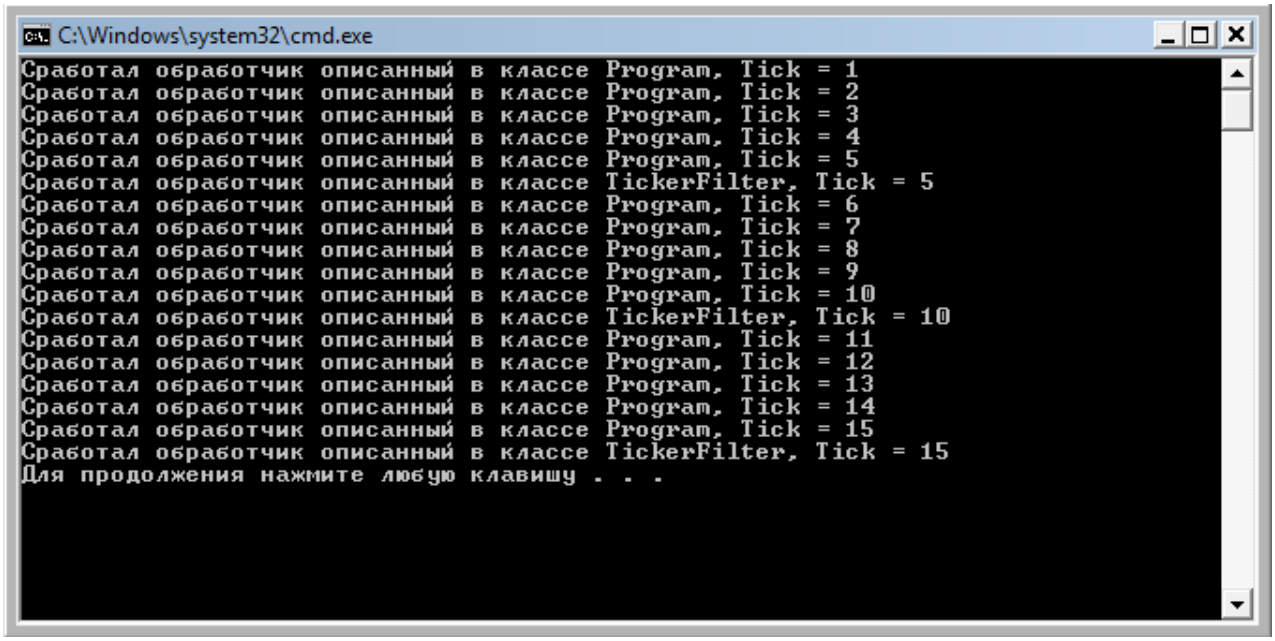
который принимает ссылку на объект вызвавший событие, и класс параметров события `TickerArgs`, который содержит значение счетчика `ticks` на момент вызова события:

```
public class TickerArgs : EventArgs
{
    public Int32 Tick;
    public TickerArgs(Int32 tick) { this.Tick = tick; }
}
```

В статическом методе `Main()` класса `Program` создается объект класса `Ticker` и объект класса `TickerFilter`, реализуется подписывание на событие и в отдельном потоке вызывается выполнение генератора сообщений (подробно создание многопоточных приложений будет рассмотрено в следующих уроках):

```
public class Program
{
    static void Main(string[] args)
    {
        Ticker ticker = new Ticker();
        ticker.TickEvent += new TickEventHandler(ticker_TickEvent);

        TickerFilter tf = new TickerFilter(ticker);

        ticker.IsEnabled = true;
        Thread thr =
            new Thread(new System.Threading.ThreadStart(ticker.RunTiker));
        thr.Start();

        Thread.Sleep(150);
        ticker.IsEnabled = false;
    }

    static void ticker_TickEvent(object sender, TickerArgs args)
    {
        Console.WriteLine(
            "Сработал обработчик описанный в классе Program, Tick = "
            + args.Tick.ToString());
    }
}

public class TickerFilter
{
    private Ticker ticker;
    public TickerFilter(Ticker ticker)
    {
        this.ticker = ticker;
        this.ticker.TickEvent += new TickEventHandler(ticker_TickEvent);
    }

    void ticker_TickEvent(object sender, TickerArgs args)
    {
        if (args.Tick % 5 == 0) Console.WriteLine(
               "Сработал обработчик описанный в классе TickerFilter, Tick = "
               + args.Tick.ToString());
    }
}
```

В результате запуска на выполнение данного примера получим:



4.5. Использование событийных средств доступа

В C# существует форма event-инструкции, которая позволяет использовать средства доступа к событиям. Эти средства доступа дают возможность управлять реализацией списка обработчиков событий. Она имеет следующий вид:

```
event nameDelegate nameEvent
{
    add {//код добавления события в цепочку событий}
    remove {//код удаления события в цепочку событий}
}
```

5. Анонимные методы

Анонимный метод – это блок кода, который применяется в качестве параметра делегата. Он имеет «заголовок», который содержит ключевое слово `delegate` и список параметров анонимного метода. Следующий код демонстрирует создание и работу анонимного метода:

```
delegate String TestAnonymousDelegate(String str);

static void Main(string[] args)
{
    String strMiddle = ", сердняя часть,";

    TestAnonymousDelegate anonymousDelegate = delegate(String parameter)
    {
        parameter += strMiddle;
        parameter += " а эта часть строки добавлена в конец.\n";
        return parameter;
    };

    Console.WriteLine(anonymousDelegate("Начало строки"));
}
```

При выполнении в консоль будет выведена строка:

```
Начало строки, сердняя часть, а эта часть строки добавлена в конец.
```

Польза от использования анонимных методов связана с экономией времени и упрощением кода. Так как не нужно создавать метод для передачи делегату, но на скорость выполнение это не влияет.

При реализации обработчиков событий также можно использовать анонимные методы. Так пример описанный в пункте 4.2. можно реализовать следующим образом:

```
static void Main(string[] args)
{
    Person person = new Person("Bill", "Gordon", new DateTime(1962, 8, 11));
    person.WorkEnded += delegate(object sender, EventArgs e)
    { Console.WriteLine("Работа выполнена !\n"); };

    person.Work();
}
```

Домашнее задание

Задание 1. Описать структуру `Article`, содержащую следующие поля: код товара; название товара; цену товара.

Описать структуру `Client` содержащую поля: код клиента; ФИО; адрес; телефон; количество заказов осуществленных клиентом; общая сумма заказов клиента.

Описать структуру `RequestItem` содержащую поля: товар; количество единиц товара.

Описать структуру `Request` содержащую поля: код заказа; клиент; дата заказа; перечень заказанных товаров; сумма заказа (реализовать вычисляемым свойством).

Задание 2. Описать перечисление `ArticleType` определяющее типы товаров, и добавить соответствующее поле в структуру `Article` из задания №1;

Описать перечисление `ClientType` определяющее важность клиента, и добавить соответствующее поле в структуру `Client` из задания №1;

Описать перечисление `PayType` определяющее форму оплаты клиентом заказа, и добавить соответствующее поле в структуру `Request` из задания №1;

Задание 3. Реализовать метод, который осуществляет поиск элемента в массиве. Метод должен принимать массив `Object[] array` , в котором должен осуществляться поиск, и делегат, определяющий, является ли элемент искомым.

Задание 4. Интерфейс `INotifyPropertyChanged` пространства имен `System.ComponentModel` определяет, что наследник содержит событие `PropertyChanged`, оповещающее об изменении свойств объекта. Изучить данный интерфейс, используя MSDN Library. Проанализировав работу общественных библиотек разработать классы: `Author, Book, Client, LibraryCard, Catalogue`. Описать их поля, поля инкапсулировать свойствами и для классов реализовать интерфейс `INotifyPropertyChanged`. Создать диаграмму классов.