Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Кубанский государственный технологический университет»

А. Г. Мурлин, В. А. Мурлина, М. В. Янаева

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Утверждено редакционно-издательским советом ФГБОУ ВО
«Кубанский государственный технологический университет»
в качестве учебного пособия

Краснодар
2021

УДК 004.4(075.8)
ББК 32.973:я73
М 913

Р е ц е н з е н т ы:

Т.П. Барановская – д-р техн. наук, проф., зав. каф. САиОИ ФГБОУ ВО «КубГАУ»;
В.Н. Марков – д-р техн. наук, проф. каф. ИСП ФГБОУ ВО «КубГТУ»;
Н.Ф. Григорьев – канд. техн. наук, руководитель отдела телекоммуникаций Краснодарского регионального информационного центра сети «КонсультантПлюс»

М 913 **Мурлин А.Г.**

Объектно-ориентированное программирование: учеб. пособие/ А.Г. Мурлин, В.А. Мурлина, М.В. Янаева. – Краснодар: Изд. ФГБОУ ВО «КубГТУ», 2021. - 151 с.

ISBN 978-5-8333-1059-5

Изложены вопросы, связанные с технологией объектно-ориентированного программирования. Пособие может быть использовано при подготовке бакалавров по направлениям 09.03.01 Информатика и вычислительная техника, 09.03.03 Прикладная информатика, 09.03.04 Программная инженерия.

УДК 004.4(075.8)
ББК 32.973:я73

ISBN 978-5-8333-1059-5

Оглавление

Введение

Основу учебного пособия «Объектно-ориентированное программирование» составляет изложение теоретического материала, большинство из которого использовалось авторами в процессе проведения учебных занятий по дисциплине «Объектно-ориентированное программирование».

Пособие предназначено для формирования навыков программирования с использованием объектно-ориентированной методологии и в качестве основного материала в рамках дисциплины «Объектно-ориентированное программирование» позволит преподавателю достичь поставленных педагогических целей согласно рабочей программе, а студентам получить прочные навыки и знания в области использования методологии объектно-ориентированного программирования.

Издание может быть использовано студентами и преподавателями высших учебных заведений как в процессе проведения занятий по программированию, так и для самостоятельного изучения соответствующей технологии. При подборе и формировании материала проработан не один десяток учебников. Из них отобраны наиболее интересные в применении на практике примеры использования объектно-ориентированного подхода к программированию.

Авторы, надеясь на оригинальность приведенных примеров, рассчитывают помочь читателям в овладении наукой и ремеслом объектно-ориентированного программирования. При подготовке материала на первый план ставилась цель – привить обучаемым навыки использования объектно-ориентированной технологии разработки программ. Учебное пособие, охватывающее базовый курс объектно-ориентированного программирования, построенное на основе языка C#, позволит студентам быстрее стать востребованными специалистами – профессионалами.

1 Принципы и основные понятия объектно-ориентированного программирования

1.1 Объектно-ориентированное программирование

Принципы объектно-ориентированного программирования (ООП) проще всего понять на примере программ моделирования. В реальном мире каждый предмет или процесс обладает набором статических и динамических характеристик, иными словами, свойствами и поведением. Поведение объекта зависит от его состояния и внешних воздействий. Например, объект «корабль» никуда не поплывет, находясь на суше, но, если повернуть штурвал, свое направление он изменит.

Объект в программировании – некоторая сущность в виртуальном пространстве, обладающая определённым состоянием и поведением, имеющая заданные значения свойств (атрибутов) и операций над ними (методов).

При создании объектно-ориентированной программы предметная область представляется в виде совокупности объектов. Выполнение программы состоит в том, что объекты обмениваются сообщениями.

При представлении реального объекта с помощью программного необходимо выделить в первом случае его существенные особенности. Их список зависит от цели моделирования. Например, объект «черепаха» с точки зрения преподавателя биологии, ветеринара или, скажем, повара будет иметь совершенно разные характеристики.

Выделение существенных с той или иной точки зрения свойств называется абстрагированием. Таким образом, программный объект – это абстракция.

Важным свойством объекта является его обособленность. Детали реализации объекта, то есть внутренние структуры данных и алгоритмы их обработки, скрыты. Для пользователя объекта детали недоступны во избежание непреднамеренных изменений. Объект используется через его интерфейс – совокупность правил доступа.

Сокрытие деталей реализации называется инкапсуляцией. В обычной жизни мы пользуемся объектами через их интерфейсы. Например, смотря телевизор, мы имеем доступ к его базовым настройкам и пакету каналов, но нюансы работы телевизора, то есть его реализация, как пользователям нам закрыты.

Таким образом, объект является «черным ящиком», замкнутым по отношению к внешнему миру. Это позволяет представить программу в укрупненном виде – на уровне объектов и их взаимосвязей, следовательно,

управлять большим объемом информации и успешно отлаживать сложные программы.

Инкапсуляция позволяет изменить реализацию объекта без модификации основной части программы, если его интерфейс остался прежним. Простота модификации является очень важным критерием качества программы, ведь любой программный продукт в течение своего жизненного цикла претерпевает множество изменений и дополнений.

Каждый год в мире пишется огромное количество новых программ, и важнейшее значение приобретает возможность многократного использования кода. Преимущество объектно-ориентированного программирования состоит в том, что для объекта можно определить наследников, корректирующих или дополняющих его поведение. При этом нет необходимости не только повторять исходный код родительского объекта, но даже иметь к нему доступ.

Наследование является мощнейшим инструментом ООП и применяется для следующих взаимосвязанных целей:

- исключения из программы повторяющихся фрагментов кода;
- упрощения модификации программы;
- упрощения создания новых программ на основе существующих.

Кроме того, только благодаря наследованию появляется возможность использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения. Наследование позволяет создавать иерархии объектов. Наследование облегчает использование библиотек объектов, поскольку программист может взять за основу объекты, разработанные кем-то другим, и создать наследников с требуемыми свойствами.

ООП позволяет писать гибкие, расширяемые и читабельные программы. Во многом это обеспечивается благодаря полиморфизму, под которым понимается возможность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа.

Инкапсуляция, наследование и полиморфизм – фундаментальные свойства, требуемые от языка, претендующего называться объектно-ориентированным. (Языки, не имеющие наследования и полиморфизма, но имеющие только классы, обычно называются основанными на классах.)

Языки объектного программирования принято делить на объектные, в которых существуют классы и объекты, и объектно-ориентированные, в которых программист может не только пользоваться предопределёнными классами, но и задавать собственные пользовательские классы.

Объектное и объектно-ориентированное программирование возникло в результате развития идеологии процедурного программирования, где

данные и подпрограммы (процедуры, функции) их обработки формально не связаны. Кроме того, в современном объектно-ориентированном программировании часто большое значение имеют понятия события (так называемое событийно-ориентированное программирование) и компонента (компонентное программирование).

Объектно-ориентированное программирование в настоящее время является абсолютным лидером в области прикладного программирования (языки Java, C#, C++, JavaScript, ActionScript и др.). В то же время в области системного программирования до сих пор лидирует парадигма процедурного программирования, и основным языком программирования является язык C.

Первым языком программирования, в котором были предложены принципы объектной ориентированности, была Симула. В момент своего появления (в 1967 году) этот язык программирования предложил поистине революционные идеи: объекты, классы, виртуальные методы и др., однако это всё не было воспринято современниками как нечто грандиозное. Тем не менее, большинство концепций были развиты Аланом Кэйем и Дэном Ингаллсом в языке Smalltalk. Именно он стал первым широко распространённым объектно-ориентированным языком программирования.

Различаются чистые и гибридные объектно-ориентированные языки. Чистые – это те, которые позволяют использовать только одну модель программирования – объектно-ориентированную. Можно объявлять классы и методы, но нельзя завести глобальные переменные и обычные функции и процедуры старого типа.

Например, Java (и его клон C#) является чистым объектно-ориентированным языком (как Eiffel и Smalltalk). На первый взгляд это кажется положительной идеей. Чистые объектно-ориентированные языки дают преимущество новичкам в объектно-ориентированном программировании, потому что программист вынужден использовать модель объектно-ориентированного программирования. C++ и Object Pascal, наоборот, – типичные примеры гибридных языков, которые позволяют программистам использовать при необходимости традиционный подход C или Pascal.

Голландская компания TIOBE Software ежемесячно обновляет мировой рейтинг языков программирования.

Рейтинг TIOBE Programming Community отражает популярность языков программирования. По нему нельзя судить о том, какой язык лучше другого – рейтинг учитывает количество специалистов по всему миру, использующих тот или иной язык, количество обучающих курсов,

сторонних поставщиков и данных поисковых систем Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube и Baidu.

Данные могут быть полезными при принятии решения о выборе языка для изучения, чтобы оставаться конкурентоспособным на рынке труда.

Взглянем на рейтинг языков программирования за июнь 2021 г., представленный на рисунке 1.

Jun 2021	Jun 2020	Change	Programming Language	Ratings	Change
1	1		C	12.54%	-4.65%
2	3	↑	Python	11.84%	+3.48%
3	2	↓	Java	11.54%	-4.56%
4	4		C++	7.36%	+1.41%
5	5		C#	4.33%	-0.40%
6	6		Visual Basic	4.01%	-0.68%
7	7		JavaScript	2.33%	+0.06%
8	8		PHP	2.21%	-0.05%
9	14	⇈	Assembly language	2.05%	+1.09%
10	10		SQL	1.88%	+0.15%
11	19	⇈	Classic Visual Basic	1.72%	+1.07%
12	31	⇈	Groovy	1.29%	+0.87%
13	13		Ruby	1.23%	+0.25%
14	9	⇊	R	1.20%	-0.99%
15	16	↑	Perl	1.18%	+0.36%
16	11	⇊	Swift	1.10%	-0.35%
17	37	⇈	Fortran	1.07%	+0.80%
18	22	⇈	Delphi/Object Pascal	1.06%	+0.47%
19	15	⇊	MATLAB	1.05%	+0.15%
20	12	⇊	Go	0.95%	-0.06%

Рис. 1. Рейтинг языков программирования

Большинство языков программирования, занимающих лидирующие позиции в рейтинге, являются объектно-ориентированными.

1.2 Классы и объекты

Класс – это обобщенное понятие, определяющее характеристики и поведение некоторого множества объектов, называемых экземплярами

класса. В программном понимании класс является типом данных, определяемым пользователем, в котором объединены структуры данных и методы их обработки.

Как описывает Павловская, класс содержит ключевое слово class, за которым следует его имя, а далее в фигурных скобках – тело класса. Кроме того, для класса можно задать его базовый класс (предок) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : предки ]
{
   тело_класса
}
```

C# позволяет указывать только один класс в качестве предка. Кроме этого, класс может реализовывать несколько интерфейсов. Более подробно схема наследования будет рассмотрена позже.

Простейший пример класса (пустого):

```
class Example
{
}
```

Спецификаторы определяют характеристики класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в таблице 1.

Т а б л и ц а 1

Значения спецификаторов

№ п/п	Спецификатор	Описание
1.	abstract	Абстрактный класс. Применяется в иерархии объектов
2.	internal	Доступ только из данного проекта (сборки)
3.	new	Задает новое описание класса взамен унаследованного от предка. Используется для вложения классов (в иерархии объектов)
4.	private	Доступ только из элементов класса, внутри которых описан данный класс. Используется для вложенных классов
5.	protected	Доступ только из данного или производного класса. Используется для вложенных классов
6.	protected internal	Доступ только из данного и производного класса в рамках текущего проекта (сборки)
7.	public	Доступ к классу не ограничен
8.	sealed	Бесплодный класс. Запрещает наследование данного класса. Применяется в иерархии объектов
9.	static	Статический класс. Позволяет обращаться к методам класса без создания экземпляра класса

Спецификаторы 2, 4–7 называются спецификаторами доступа. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут комбинироваться с остальными спецификаторами, но не могут комбинироваться между собой.

Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется вложенным. В зависимости от места описания класса некоторые из этих спецификаторов могут быть запрещены. В данном учебном пособии рассматриваются классы, которые описываются непосредственно в пространстве имен. Для таких классов, говорит нам Шилдт, допускаются только два спецификатора: public и internal. Если ни один спецификатор доступа не указан, то по умолчанию используется спецификатор internal.

Объекты (экземпляры класса) создаются явным образом с помощью операции new, например:

```
Example a = new Example (); // Создается экземпляр класса Example
```

Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его члены – данные и методы. В классе могут присутствовать статические члены, которые существуют в единственном экземпляре для всех объектов класса. Статические данные часто называют данными класса, а остальные – данными экземпляра. Для работы с данными класса используются статические методы класса, для работы с данными экземпляра – методы экземпляра, или просто методы.

В общем случае класс может содержать следующие элементы:

– данные, являющиеся переменными или константами;

– методы, реализующие не только вычисления, но и другие действия с классом или его экземпляром;

– конструкторы, реализующие действия по инициализации экземпляров класса или класса в целом;

– деструкторы, определяющие действия, которые необходимо выполнить непосредственно перед уничтожением объекта;

– свойства, определяющие возможности доступа к членам класса;

– индексаторы, обеспечивающие возможность доступа к членам класса по их порядковому номеру (индексу);

– операции, задающие действия с экземплярами класса с помощью знаков операций;

– события, определяющие уведомления, которые может генерировать класс;

– типы или структуры данных, определенные внутри класса.

Прежде чем приступить к проектированию классов, отметим, что классы относятся к ссылочным типам данных. Принципиальное различие между размерными и ссылочными типами состоит в способе хранения их значений в памяти. Для размерных типов, таких как int, bool и т.п., фактическое значение хранится в стеке (или как часть большого объекта ссылочного типа). Адрес переменной ссылочного типа тоже хранится в стеке, но сам объект хранится в куче (динамической памяти).

Данное различие существенно скажется на выполнении операций присваивания и сравнения объектов. Сам механизм выполнения присваивания один и тот же для величин любого типа как ссылочного, так и размерного, однако результаты различаются. При присваивании значения копируется значение, а при присваивании ссылки – ссылка.

Например, пусть были созданы три объекта: a, b и c, а затем выполнено присваивание b = c. Теперь ссылки b и c указывают на один и тот же объект. Старое значение b становится недоступным и удаляется сборщиком мусора.

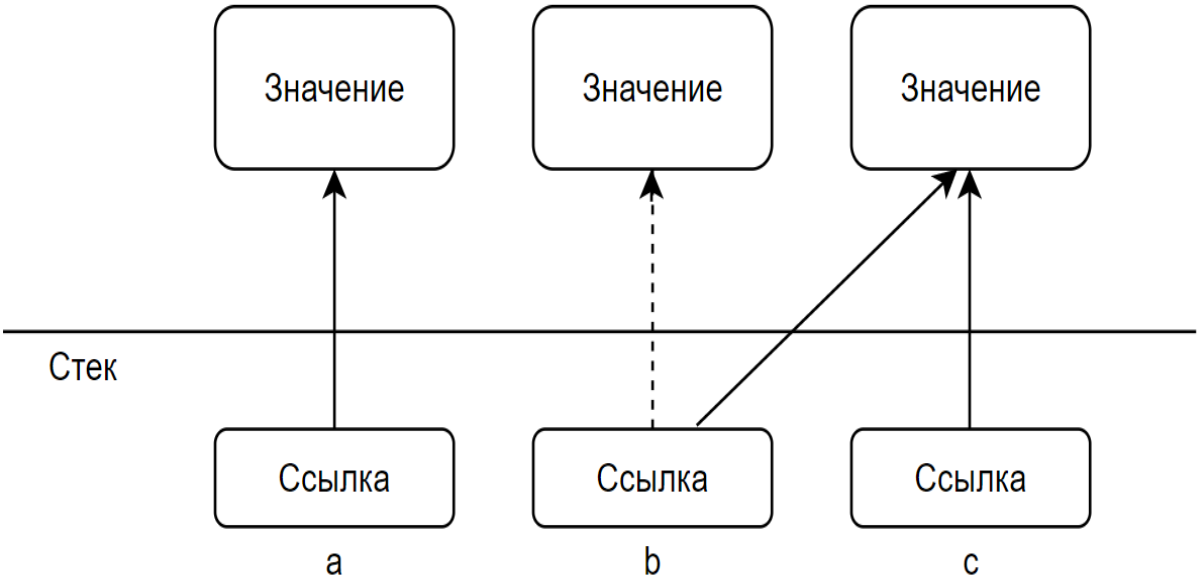

Рис. 2. Присваивание объектов

Рассмотрим теперь операцию сравнения. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одну и ту же область памяти. Так, объекты b и c равны, т.к. они ссылаются на одну и ту же область памяти, но a не равно b даже при равенстве их значений.

1.3 Данные

Члены-данные класса могут быть переменными или константами и задаются в соответствии с правилами объявления идентификаторов. Синтаксис описания членов-данных:

[атрибуты] [спецификаторы] [const] тип имя [= начальное_значение];

Возможные спецификаторы для данных приведены в таблице 2.

Т а б л и ц а 2

Спецификаторы для данных

№ п/п	Спецификатор	Описание
1.	internal	Доступ только из данной сборки
2.	new	Новое описание поля, скрывающее унаследованный элемент класса
3.	private	Доступ только из данного класса
4.	protected	Доступ только из данного и производных классов
5.	protected internal	Доступ только из данного и производных классов из данной сборки
6.	public	Доступ к элементу не ограничен
7.	readonly	Поле доступно только для чтения (значения таких полей можно установить либо при описании, либо в конструкторе)
8.	static	Одно поле для всех экземпляров класса
9.	volatile	Поле может изменяться другим процессом или системой

Для констант можно использовать только спецификаторы 1–6.

По умолчанию члены-данные считаются закрытыми, т.е. для них установлен спецификатор private. Для членов-данных этот вид доступа является предпочтительным, поскольку поля определяют внутреннее строение класса, которое должно быть скрыто от пользователя. Все методы класса имеют непосредственный доступ к его закрытым полям.

Поля, описанные со спецификатором static, а также константы существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса. Обращение к полю класса выполняется с помощью операции доступа (точка). Справа от точки задается имя поля, слева – имя экземпляра для обычных полей или имя класса для статических.

Теперь рассмотрим пример разработки класса и возможные способы обращения к его полям. Но перед этим создадим проект в Visual Studio.

Запускаем Visual Studio и нажимаем на «Создание проекта».

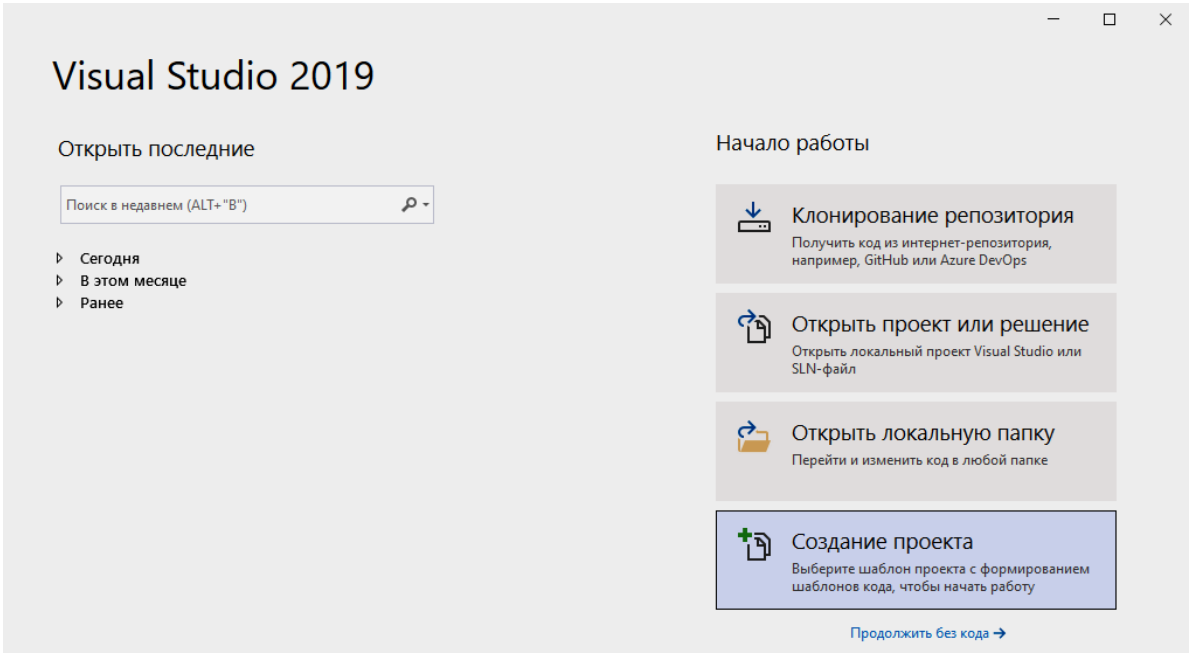

Рис. 3. Создание проекта

1. В открывшемся разделе выбираем «Консольное приложение (.NET Core)».

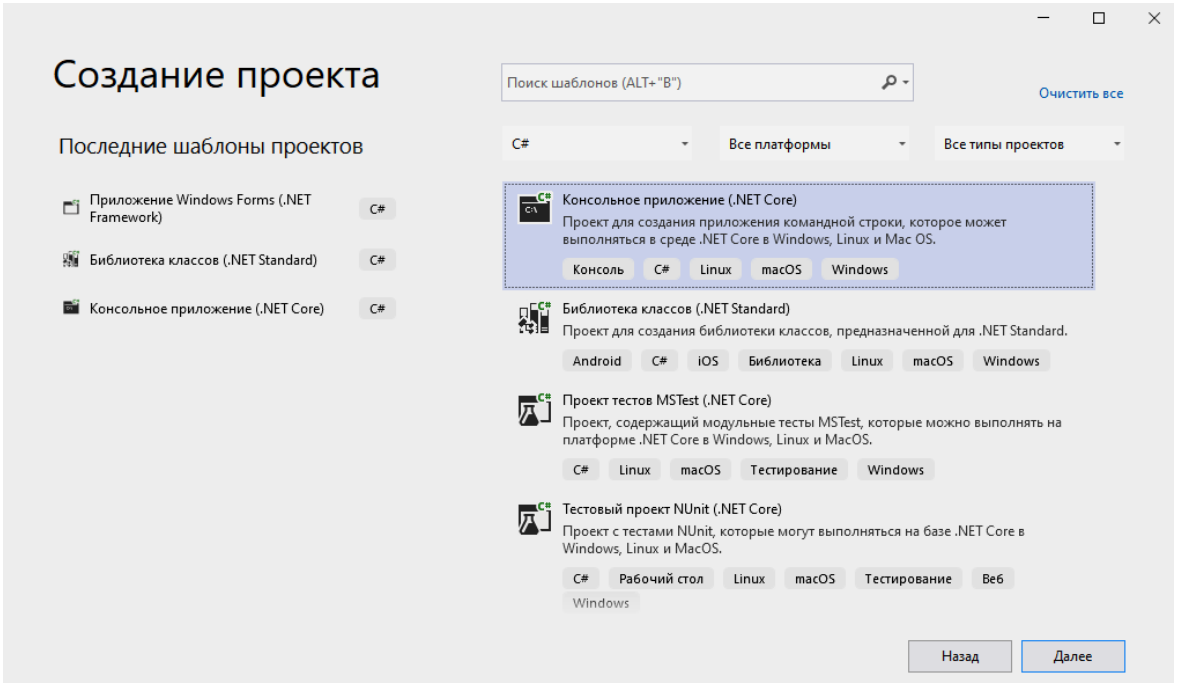

Рис. 4. Создание консольного приложения

2. Редактируем по желанию название и расположение проекта, щелкаем «Создать».

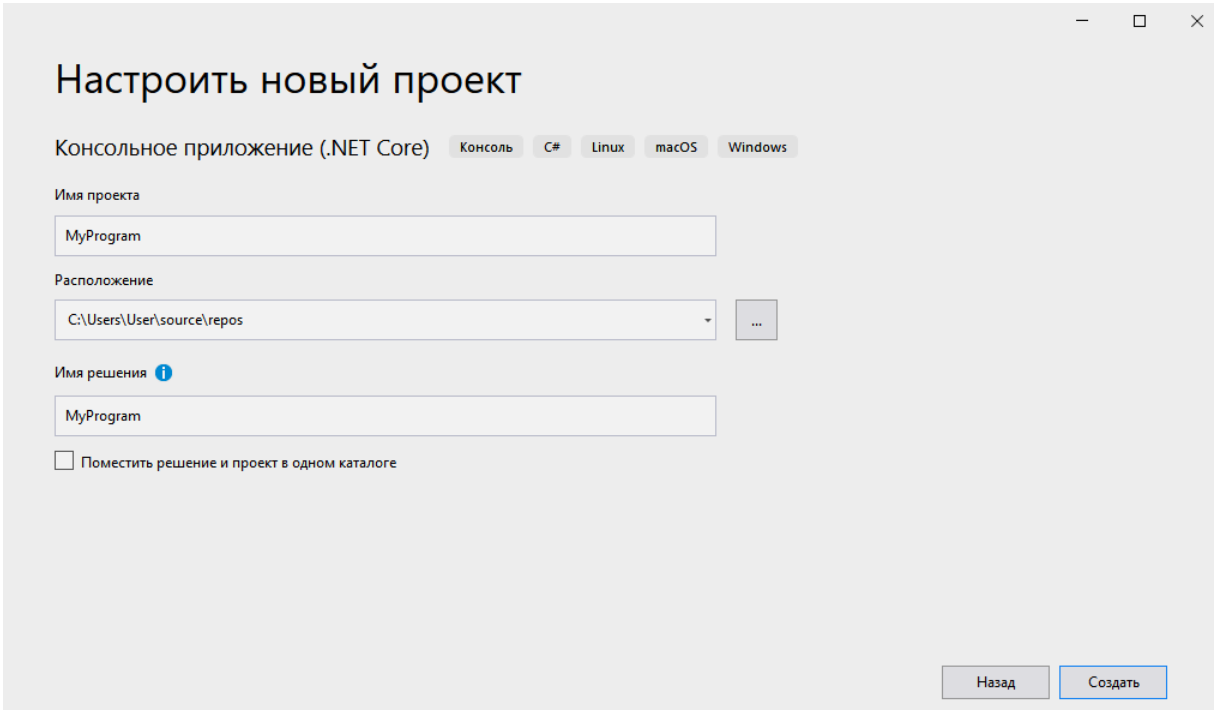

Рис. 5. Настройка нового проекта

3. Готово. Теперь можно приступать к написанию кода.

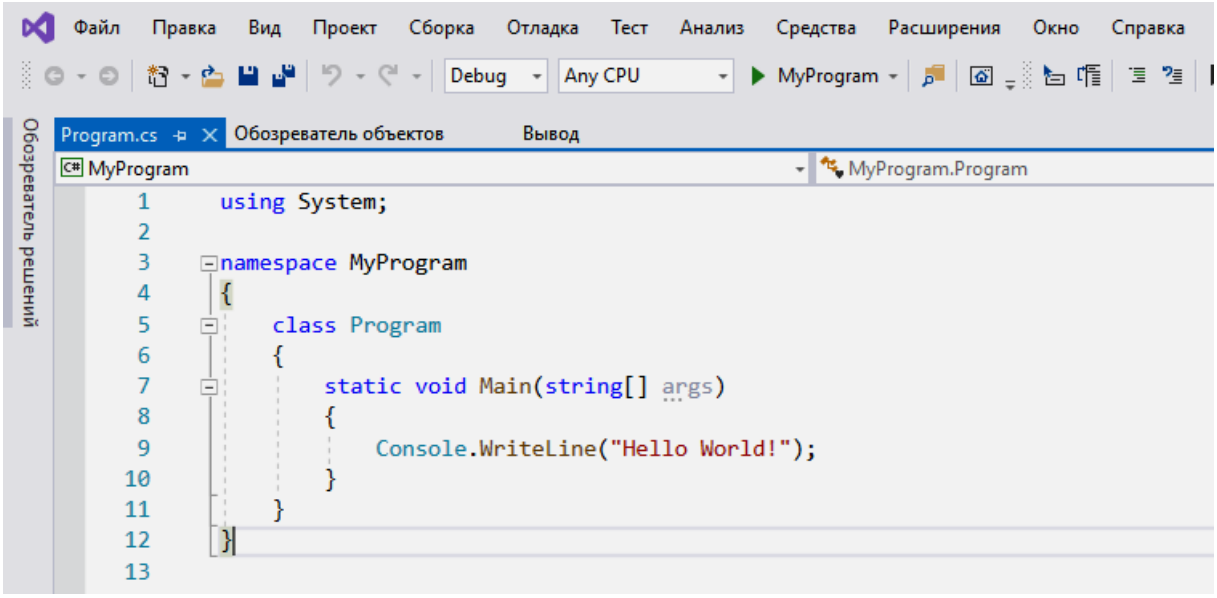

Рис. 6. Открытый проект

Далее будет рассмотрена программа, включающая в себя класс Circle, который описывает окружность (ее координаты, радиус и т.д.).

```
using System;

namespace MyProgram
{
   class Circle
   {
public int x=0;
public int y=0;
public int radius=3;
public const double pi = 3.14;
public static readonly string name = "Окружность";
double area;
   }
   class Program
  {
     static void Main()
     {
Circle oneCircle = new Circle(); // Создание экземпляра класса
Console.WriteLine("pi={0}", Circle.pi); // Обращение к константе
// Обращение к статическому полю
Console.WriteLine("Используется объект {0}", Circle.name);
// Обращение к обычным полям
Console.WriteLine("Центр в точке ({0},{1}), радиус {2}", oneCircle.x,
oneCircle.y, oneCircle.radius);
oneCircle.radius = 100;
Console.WriteLine(" Новая окружность с центром в точке ({0},{1}) и
радиусом {2}", oneCircle.x, oneCircle.y, oneCircle.radius);

        // Так как поле area имеет спецификатор private, то обращение к
        // нему, наподобие следующего, вызовет ошибку:
        // oneCircle.area = 2 * Circle.pi * oneCircle.radius;
        // Попытка изменить значение поля name также вызовет ошибку, т.к.
        // это поле доступно только для чтения:
        // Circle.name="квадрат";
     }
  }
}
```

Результат работы программы:

Используется объект Окружность
Центр в точке (0, 0), радиус 3
Новая окружность с центром в точке (0, 0) и радиусом 100

1.4 Методы

Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами одного класса совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший метод экземпляр.
В явном виде параметр this применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

```
using System;

namespace MyProgram
{
    class Circle
    {
        public int x=0;
        public int y=0;
        public int radius=3;
        public const double pi = 3.14;
        public static readonly string name = "Окружность";

        public void Set (int x, int y, int radius)
        {
            // Использует параметр this для обращения к полям класса,
            // т.к. их имена совпадают с именами параметров метода.
            this.x = x;
            this.y = y;
            this.radius= radius;
         }

         public void Show()
        {
            Console.WriteLine("{0} с центром в точке ({1},{2}) радиусом
{3}", name, x, y, radius);
```

```
        }
    }


    class Program
    {
        static void Main()
        {
            Circle oneCircle = new Circle();
            oneCircle.Show();
            oneCircle.Set(1, 1, 100);
            oneCircle.Show();
        }
    }
}
```

Результат работы программы:
Окружность с центром в точке (0, 0) радиусом 3
Окружность с центром в точке (1, 1) радиусом 100

С добавлением новых классов в программу резко увеличивается ее размер, что затрудняет ее прочтение. Поэтому следует руководствоваться одним простым принципом «один класс – один файл».

Создание нового файла для класса приводится ниже.

1. В верхней вкладке «Проект» выбрать «Добавить класс».

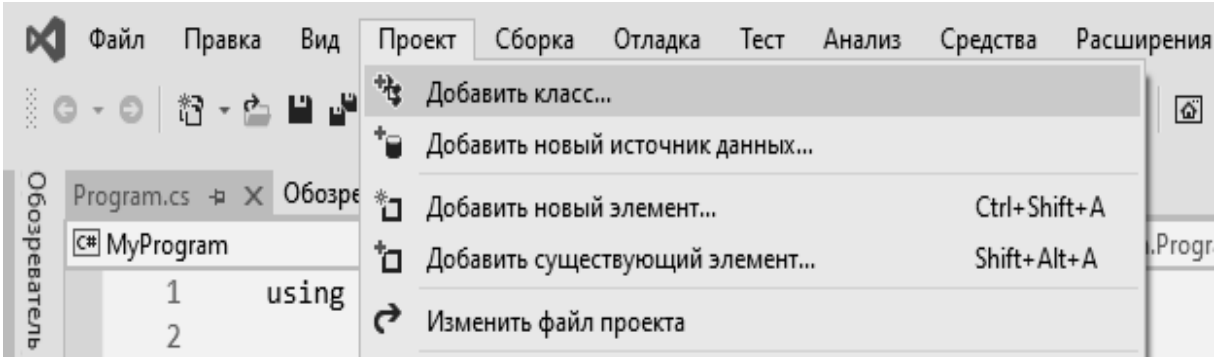

Рис. 7. Добавление класса в проект

2. В открывшемся окне ввести *Имя* класса, нажать кнопку «Добавить».

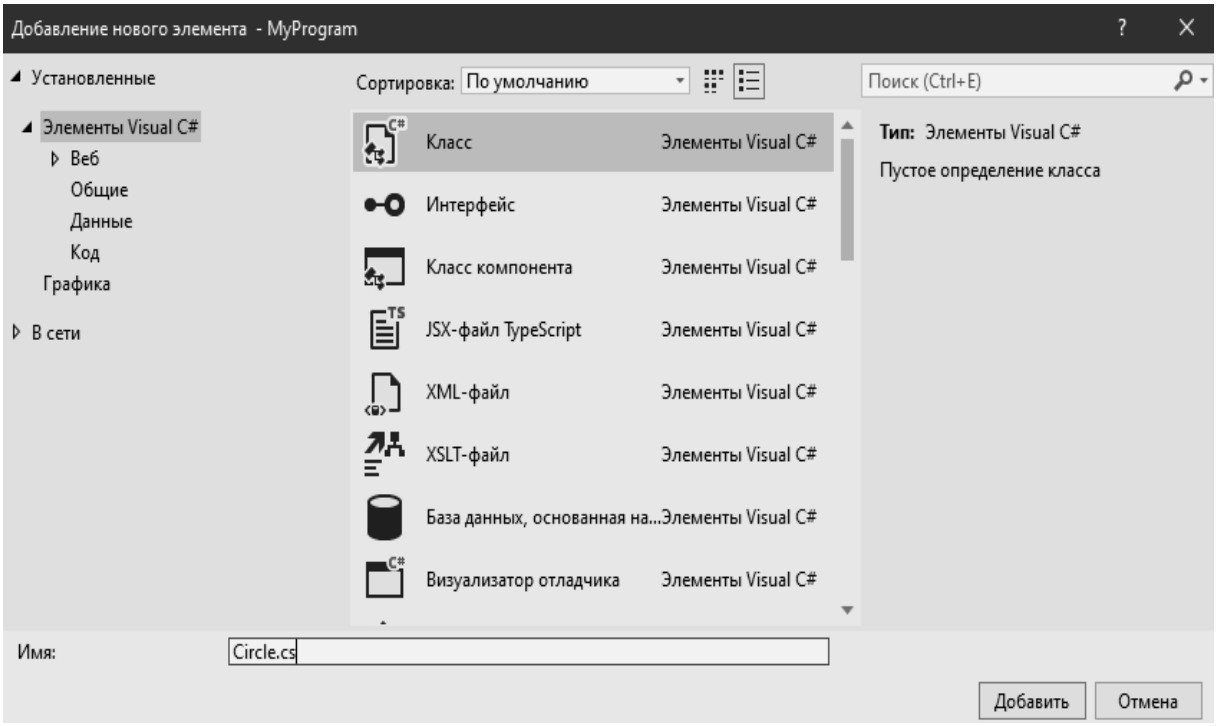

Рис. 8. Выбор имени для нового класса

3. Готово, теперь сюда можно перенести класс, ранее созданный в Program.cs.

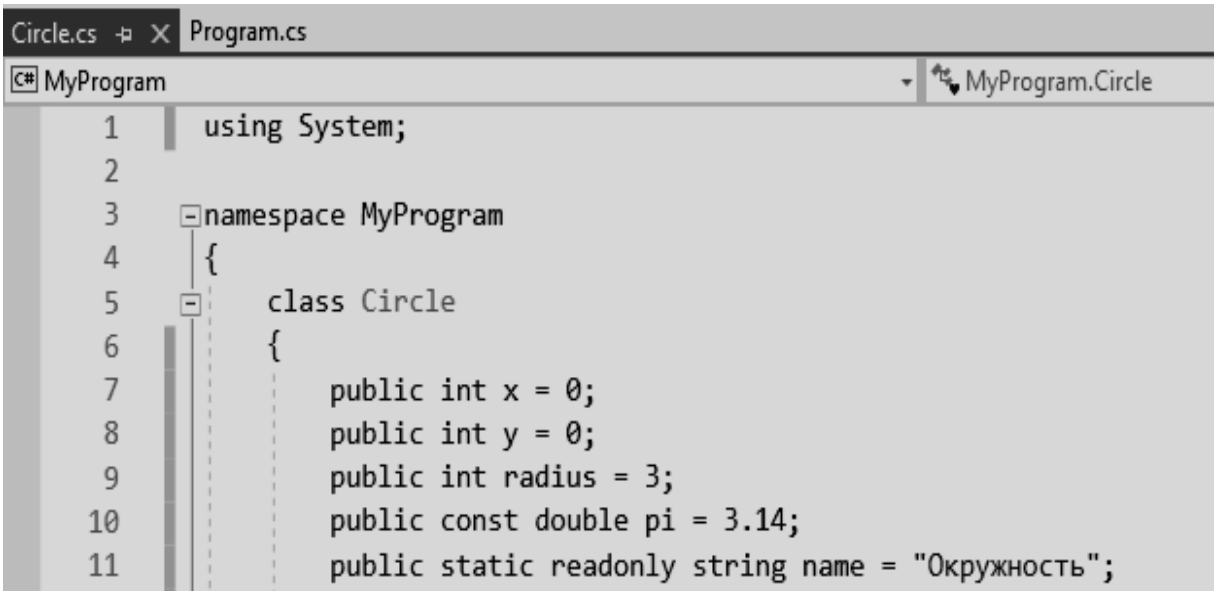

Рис. 9. Перенесенный класс Circle

1.5 Конструкторы

Конструктор предназначен для инициализации объекта. Конструкторы класса делятся на статические конструкторы и конструкторы экземпляра.

Конструктор экземпляра вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса. Рассмотрим основные свойства конструкторов:

1. Конструктор не возвращает значение, даже типа void.

2. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации, выбор конструктора происходит автоматически, в зависимости от количества и типа передаваемых параметров.

3. Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, то полям значимых типов присваивается нуль, полям ссылочных типов – значение null.

До сих пор задавали начальные значения полей класса при описании класса. Это удобно в том случае, когда для всех экземпляров класса начальные значения полей одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать с помощью явного задания конструктора.

Добавим в класс Circle два конструктора: первый – инициализирует только поле radius, второй – поля x, y, radius.

```
namespace MyProgram
{
   class Circle
  {
      public int x;
      public int y;
      public int radius;
      public static readonly string name = "Окружность";

      public Circle(int radius) //конструктор 1- инициализирует только поле radius
        {
          this.radius= radius;
        }

        // Конструктор 2 - инициализирует поля x, y, radius
        public Circle (int x, int y, int radius)
```

```
        {
            this.x = x;
            this.y = y; this.
            radius= radius;
        }

        public void Set(int x, int y, int radius)
        {
            this.x = x;
            this.y = y;
            this.radius= radius;
        }

        public void Show()
        {
            Console.WriteLine("{0} с центром в точке ({1},{2}), радиусом
{3}", name, x, y, radius);
        }
    }
}
```

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-то, то можно воспользоваться параметром this для того, чтобы вызвать один конструктор из другого. Например:

```
public Circle( int radius) //конструктор 1
{
  this radius= radius;
}

public Circle (int x, int y, int radius) :this (radius) //конструктор 2
{
  this.x = x;
  this.y = y;
}
```

В данном случае конструктор 2 вызывает конструктор 1. Запись вида :this(radius) называется инициализатором, то есть кодом, который будет выполнен до начала выполнения тела конструктора 2.

Рассмотрим, как вызвать конструкторы класса Circle из класса Program:

```
using System;

namespace MyProgram
{
  class Program
 {
    static void Main()
   {
      Circle oneCircle = new Circle(1); //вызов конструктора 1
      oneCircle.Show();
      Circle twoCircle=new Circle(1, 1, 100); //вызов конструктора 2
      twoCircle.Show();
    }
 }
}
```

Результат работы программы:
Окружность с центром в точке (0, 0) радиусом 1
Окружность с центром в точке (1, 1) радиусом 100

1.6 Деструкторы

В C# существует специальный вид метода, называемый деструктором, который вызывается сборщиком мусора непосредственно перед удалением объекта из памяти.

Напоминаем, что сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени.

В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта. Например, проверяется все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т. п.). Если ваш класс не используется для обеспечения доступа к какому-либо системному ресурсу (например, объекты графического контекста рисования), использование деструкторов настоятельно не рекомендуется.

Синтаксис деструктора:

```
[атрибуты] [extern] ~имя_класса()
{
  тело_деструктора
}
```

Деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой (~), символизирующей обратные по отношению к конструктору действия. Тело деструктора представляет собой блок или просто точку с запятой. Добавим в класс Circle следующий деструктор:

```
~Circle()
{
  Console.WriteLine("Удалена {0} с центром в точке ({1},{2}), радиусом {3}", name, x, y, radius);
}
```

Теперь посмотрим, как выполнится следующая программа:

```
using System;

namespace MyProgram
{
  class Program
  {
    static void Main()
    {
      Circle oneCircle = new Circle(1);
      oneCircle.Show();
      Circle twoCircle=new Circle(1, 1, 100);
      twoCircle.Show();
      Circle threeCircle = new Circle(5, 2, 10);
      threeCircle.Show();
    }
  }
}
```

Результат работы программы:
Окружность с центром в точке (0, 0) радиусом 1
Окружность с центром в точке (1, 1) радиусом 100
Окружность с центром в точке (5, 2) радиусом 10
Удалена Окружность с центром в точке (5, 2) радиусом 10
Удалена Окружность с центром в точке (1, 1) радиусом 100
Удалена Окружность с центром в точке (0, 0) радиусом 1
Обратите внимание на то, что деструкторы были вызваны автоматически.

Вернемся к проблеме потерянных ссылок на объекты:

```
using System;

namespace MyProgram
{
  class Program
  {
    static void Main()
    {
      //создаем три объекта
      Circle oneCircle = new Circle(1);
      Circle twoCircle=new Circle(1, 1, 100);
      Circle threeCircle = new Circle(5, 2, 10);
     //выводим информацию об объектах
      oneCircle.Show();
      twoCircle.Show();
      threeCircle.Show();
      Console.WriteLine();
     // теперь threeCircle и oneCircle ссылаются на один и тот же объект, //к объекту, представляющему окружность с центром (0, 0) и радиусом 1 не //ведет ни одной ссылки
      threeCircle=oneCircle;
     //изменяем значения по ссылке threeCircle
      threeCircle.Set(-1, -1, 20);
     //вновь выводим информацию об объектах
     oneCircle.Show(); twoCircle.Show();
     threeCircle.Show();
     Console.WriteLine();
    }
  }
}
```

Результат работы программы:

```
Окружность с центром в точке (0, 0) радиусом 1
Окружность с центром в точке (1, 1) радиусом 100
Окружность с центром в точке (5, 2) радиусом 10
Окружность с центром в точке (-1, -1) радиусом 20
Окружность с центром в точке (1, 1) радиусом 100
Окружность с центром в точке (-1, -1) радиусом 20
Удалена Окружность с центром в точке (5, 2) радиусом 10
Удалена Окружность с центром в точке (1, 1) радиусом 100
Удалена Окружность с центром в точке (-1, -1) радиусом 20
```

Данный пример показывает, что ссылки oneCircle и threeCircle действительно ссылаются на один и тот же объект. А ссылка на окружность с центром (0, 0) и радиусом 1 потеряна. Однако сборщик мусора удалит все объекты, в том числе и те, ссылки на которые были потеряны.

1.7 Свойства

Как уже отмечалось ранее, данные по умолчанию считаются закрытыми, т.е. для них установлен спецификатор private. Для полей этот вид доступа является предпочтительным, поскольку данные определяют внутреннее строение класса, которое должно быть скрыто от пользователя.

Для того чтобы определить для пользователя способ доступа к полям класса, можно использовать такой функциональный элемент класса как свойство. Свойство позволяет сделать поле доступным только для чтения или только для записи (при этом можно проверить допустимость присваиваемых полю значений).

Синтаксис свойства:

```
[атрибуты] [спецификаторы] тип имя_свойства
{
   [get { код_доступа_для_чтения} ]
   [set { код_доступа_для записи} ]
}
```

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые, т.е со спецификатором public.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или при установке (set) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, то свойство доступно только для чтения. Если отсутствует часть get, то свойство доступно только для записи. Блок get должен содержать оператор return, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства.

Обратится к свойству для получения значения можно, например, следующим образом:

x = имя_класса.имя_свойства;

Обратится к свойству для установки значения можно, например, следующим образом:

имя_класса.имя_свойства = значение;

При этом *значение* будет записано в параметр value, который будет использоваться в свойстве для установки соответствующего значения поля. При необходимости можно осуществлять проверку допустимости устанавливаемых значений.

Внесем следующие изменения в класс Circle:

– сделаем поля x, y, radius, name доступными только для чтения;

– создадим свойства для работы с закрытыми полями x, y, radius, позволяющие получать и устанавливать значения соответствующих полей; для поля radius в блоке set будем проводить проверку допустимости устанавливаемого значения;

– создадим свойство для работы с полем name, доступным только для чтения;

– в конструкторе 1 инициализацию поля radius будем проводить через соответствующее свойство для проверки допустимости устанавливаемого значения;

– создадим свойство, доступное только для чтения, позволяющее определять площадь круга.

```
using System;

namespace MyProgram
{
  class Circle
  {
    //теперь поля закрыты и доступ из другого класса к ним
невозможен
    private int x;
    private int y;
    private int radius;
    static readonly string name="Окружность";

    public Circle(int radius)
    {
      //устанавливаем поле radius через одноименное свойство, для
//проверки  допустимости устанавливаемого значения
      Radius= radius;
    }
    public Circle (int x, int y, int radius) :this (radius)
    {
      this.x = x;
      this.y = y;
    }
```

```
        public void Show()
        {
          Console.WriteLine("{0} с центром в точке ({1},{2}), радиусом
{3}", name, x, y, radius);
        }

        public int X //свойство для обращения к закрытому полю x
        {
          get
          {
            return x;
          }
          set
          {
            x = value;
          }
        }

        public int Y //свойство для обращения к полю y
        {
          get
          {
            return y;
          }
          set
          {
            y = value;
          }
        }

        public int Radius //свойство для обращения к полю radius
        {
          get
          {
            return radius;
          }
          set
          {
            if (value>0)
            {
              radius = value;
```

```
            }
            else
            {
              radius=0;
              Console.WriteLine("Недопустимое значение для радиуса
окружности {0}", value);
            }
          }
        }

        public string Name //свойство для работы с закрытым readonly полем
        {
          get
          {
            return name;
          }
        }

        public double Area // свойство, доступное только для чтения
        {
          get
          {
            return Math.PI*radius*radius;
          }
        }
      }
    }
```

Таким образом, свойство Area не пытается получить или установить значение какого-либо закрытого поля класса, оно решает самостоятельную задачу – вычисляет площадь круга. Поэтому свойство можно использовать как разновидность метода. Данный пример при обнаружении ошибочных данных выводит сообщение напрямую на консоль. В реальных приложениях это недопустимо. Для сообщения о возникновении ошибки рекомендуется применять механизм обработки исключений.

Рассмотрим на примере использование разработанных свойств:

```
using System;
namespace MyProgram
{
  class Program
  {
    static void Main()
```

```
        {
          Circle oneCircle = new Circle(-1);
          oneCircle.Show();
          Console.WriteLine();
          oneCircle.X=-1; // параметр value принимает значение -1
          oneCircle.Y=1; // параметр value принимает значение 1
          oneCircle.Radius=2; // параметр value принимает значение 2
          oneCircle.Show();
          Console.WriteLine("Заданная {0} имеет площадь {1:f}",
oneCircle.Name, oneCircle.Area);
        }
      }
    }
```

Результат работы программы:
Недопустимое значение для радиуса окружности -1
Окружность с центром в точке (0, 0) радиусом 0
Окружность с центром в точке (-1, 1) радиусом 2
Заданная Окружность имеет площадь 12.57

В .NET Framework 2.0 появилась возможность задавать отдельные права доступа для каждого из блоков set и get. Например, если для каких-либо действий необходимо задавать поле Area принудительно (изменяя при этом радиус), то не нужно, чтобы это мог делать пользователь, можно использовать следующую конструкцию:

```
public double Area
{
  get
 {
    return Math.PI * radius * radius;
  }
  private set
  {
     r = Math.sqrt(value / (Math.PI));
  }
}
```

Начиная с версии .NET Framework 3.0 можно использовать упрощенный синтаксис для свойств, не реализующих никакой логики. Например, вместо

```
private int y;
public int Y //свойство для обращения к полю y
```

```
{
  get
  {
   return y;
  }
  set
  {
   y = value;
  }
}
```

Можно писать просто:
public int Y { get; set; }

Среда исполнения сама, в процессе компиляции создаст необходимое поле и пропишет код для его изменения.

1.8 Индексаторы

Индексатор представляет собой разновидность свойства и позволяет организовать доступ к скрытым полям класса по индексу, например так же, как мы обращаемся к элементу массива. Троелсен указывает, что синтаксис индексатора аналогичен синтаксису свойства:

```
// последние [ ] являются обязательными элементами синтаксиса
[атрибуты] [спецификаторы] тип this [список параметров]
{
   [get { код_доступа} ]
   [set { код_доступа }]
}
```

Спецификаторы аналогичны спецификаторам свойств и методов. Чаще всего индексаторы объявляются со спецификатором public.

Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке (set) значения некоторого элемента класса. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, индексатор доступен только для чтения, если отсутствует часть get, индексатор доступен только для записи.

В качестве примера рассмотрим индексатор, который позволяет осуществить доступ к координатам центра окружности: индекс 1 соответствует координате x, индекс 2 – координате y:

```
public int this[int i]
{
   get
   {
      if (i==1) return x;
      else if (i==2) return y;
            else
            {
               Console.WriteLine("Недопустимый индекс");
               return 0;
            }
   }
    set
   {
      if (i==1) x=value;
      else if (i==2) y=value;
            else Console.WriteLine("Недопустимый индекс");
   }
}
```

Использовать индексатор можно следующим образом:

```
Circle oneCircle = new Circle(1);
oneCircle[1]=4;
oneCircle[2]=5;
Console.WriteLine("Центр окружности ({0}, {1})", oneCircle[1], oneCircle[2]);
```

Результат работы фрагмента программы:

Центр окружности (4, 5)

Пример, который мы рассмотрели, носит учебный характер. На практике индексаторы очень удобно применять для создания специализированных массивов, на которые накладываются какие-либо ограничения. Рассмотрим в качестве примера класс массив, значения элементов которого находятся в диапазоне [0, 100].

```
using System;
namespace MyProgram
```

```
{
   class DemoArray
  {
     int[] MyArray; //закрытый массив
//конструктор, использующий переменное количество аргументов
     public DemoArray(params int []array)
    {
       MyArray = new int[array.Length];
       Array.Copy(array,MyArray, array.Length);
    }

    public void Show()
   {
      foreach (int item in MyArray)
     {
        Console.Write("{0} ", item);
     }
     Console.WriteLine();
   }

    public int this[int i] //индексатор
   {
      get
     {
        if (i = MyArray.Length)
       {
          Console.WriteLine("Индекс {0} выходит за границы
массива", i);
          return 0;
       }
       else return MyArray[i];
     }
     set
    {
       if (i = MyArray.Length) Console.WriteLine("Индекс {0}
выходит за границы массива", i);
        else if (value >= 0 && value <= 100) MyArray[i] = value;
            else Console.WriteLine("Присваивается недопустимое
значение {0}", value);
     }
   }
```

```
  }
}
```

Использовать индексатор можно следующим образом:

```
using System;

namespace MyProgram
{
   class Program
  {
     static void Main()
    {
       DemoArray ob=new DemoArray(2, 6, 7, 8, 1, 3, 0, 8, 4);
       ob.Show();
       ob[2]=100;
       ob[3]=200;
       ob[100]=10;
       ob.Show();
    }
  }
}
```

Результат работы программы:
2 6 7 8 1 3 0 8 4
Присваивается недопустимое значение 200
Индекс 100 выходит за границы массива
2 6 100 8 1 3 0 8 4

Язык C# допускает использование многомерных индексаторов, которые могут использоваться, например, для многомерных массивов. Синтаксис описания многомерного индексатора:

```
public int this[int i, int j, …] // многомерный индексатор
{
   get
  { }
   set
  { }
}
```

Аналогичным образом в качестве параметров индексатора можно использовать не только целые числа, но и строки или любые другие программные объекты.

Ниже можно рассмотреть пример, включающий два индексатора, один из которых принимает в аргументах строковое значение.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace ExampleIndexators
{
   class Student
   {
      string name;
      string surname;
      string middleName;
      string uniqueId;

      // Индексатор для получения или установки значений полей
      // данного студента (name, surname, middleName, uniqueId)
      public string this[string prop]
      {
         get
         {
            switch(prop)
            {
               case "Имя": return name;
               case "Фамилия": return surname;
               case "Группа": return middleName;
               case "Уникальный номер": return uniqueId;
               default: return null;
            }
         }

         set
         {
            switch (prop)
            {
               case "Имя":
```

```
                name = value;
                break;
            case "Фамилия":
                surname = value;
                break;
            case "Отчество":
                middleName = value;
                break;
            case "Уникальный номер":
                uniqueId = value;
                break;
        }
    }
}

public override string ToString()
{
    return $"{name} {surname} ({middleName}) - {uniqueId}";
}
}

class StudentGroup
{
List<Student> students = new List<Student>();
public int Count
{
    get
    {
        return students.Count;
    }
}

// Индексатор для получения или установки экземпляров
// студентов в списке students
public Student this[int i]
{
    get
    {
        if (i > Count) return null;
        if (i == Count)
        {
```

```
                // Если i на 1 больше, чем макс. индекс students,
                // добавляем в students нового студента
                students.Add(new Student());
            }
            return students[i];
        }
        set
        {
            if (i > Count) return;
            if (i == Count)
                students.Add(value);
            else
                students[i] = value;
        }
    }

    public override string ToString()
    {
        string text = "Группа:\n";
        for (int i = 0; i < Count; i++)
            text += $"{i+1}) {students[i]}\n";

        return text;
    }
}

public partial class Form1 : Form
{
    StudentGroup studentGroup = new StudentGroup();

    public Form1()
    {
        InitializeComponent();
    }
    //По нажатию кнопки – сохранение изменений в группе студентов
    private void saveChangesButton_Click(object sender, System.EventArgs e)
    {
        // Создаем новый экземпляр группы студентов
        studentGroup = new StudentGroup();
```

```
            // Проходим все ячейки таблицы dataGridView1, каждая строка – отдельный студент
            foreach (DataGridViewRow row in dataGridView1.Rows) {
                Student student = new Student();
                bool isEmptyFields = false;

                foreach (DataGridViewCell cell in row.Cells)
                {
                    string value = (string)dataGridView1[cell.ColumnIndex, row.Index].Value;
                    // Если значение ячейки пустое, значит не заполнены все
        // данные студента,  и всю строку (т.е. студента) не берем в счет
                    if (value == null)
                    {
                        isEmptyFields = true;
                        break;
                    }

                    // columnHeader содержит название поля, передаваемое в
//качестве аргумента в индексатор для установки данных студента
                    string columnHeader = dataGridView1.Columns[cell.ColumnIndex].HeaderText;
                    student[columnHeader] = value;
                }

                if (!isEmptyFields)
                    studentGroup[studentGroup.Count] = student;
            }
        }


        // По нажатию кнопки – отображение текущего списка студентов через MessageBox
        private void showStudentsButton_Click(object sender, System.EventArgs e)
        {
            MessageBox.Show(studentGroup.ToString());
        }
    }

    …
```

}

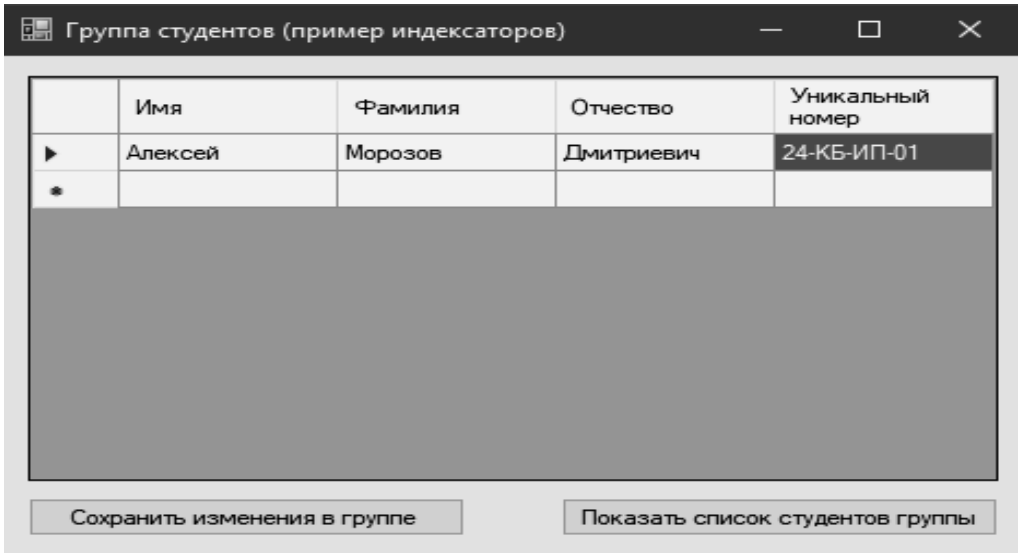

Рис. 10. Таблица со списком студентов

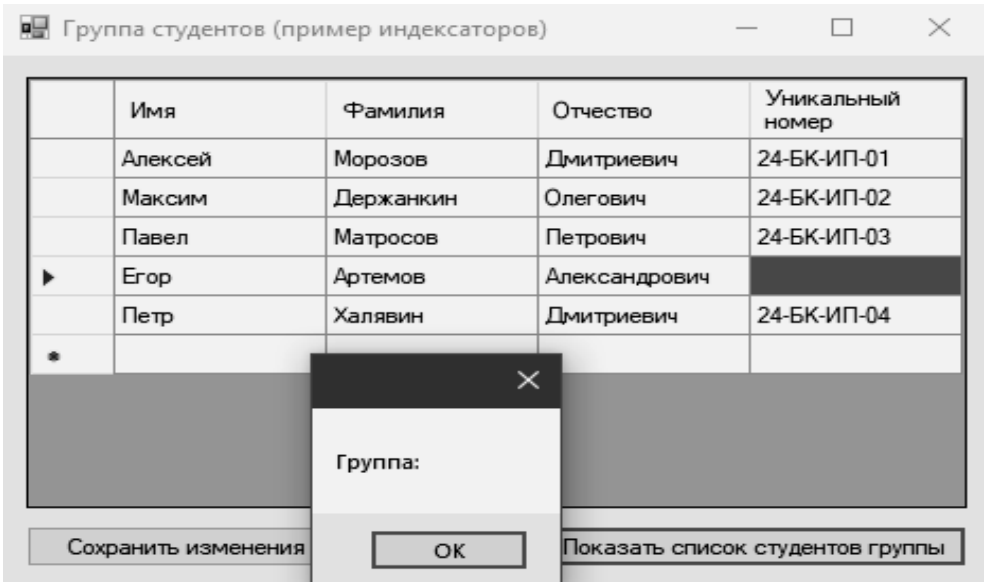

Рис. 11. Не сохранены изменения в группе студентов, поэтому при выводе списка студентов он оказывается пустым

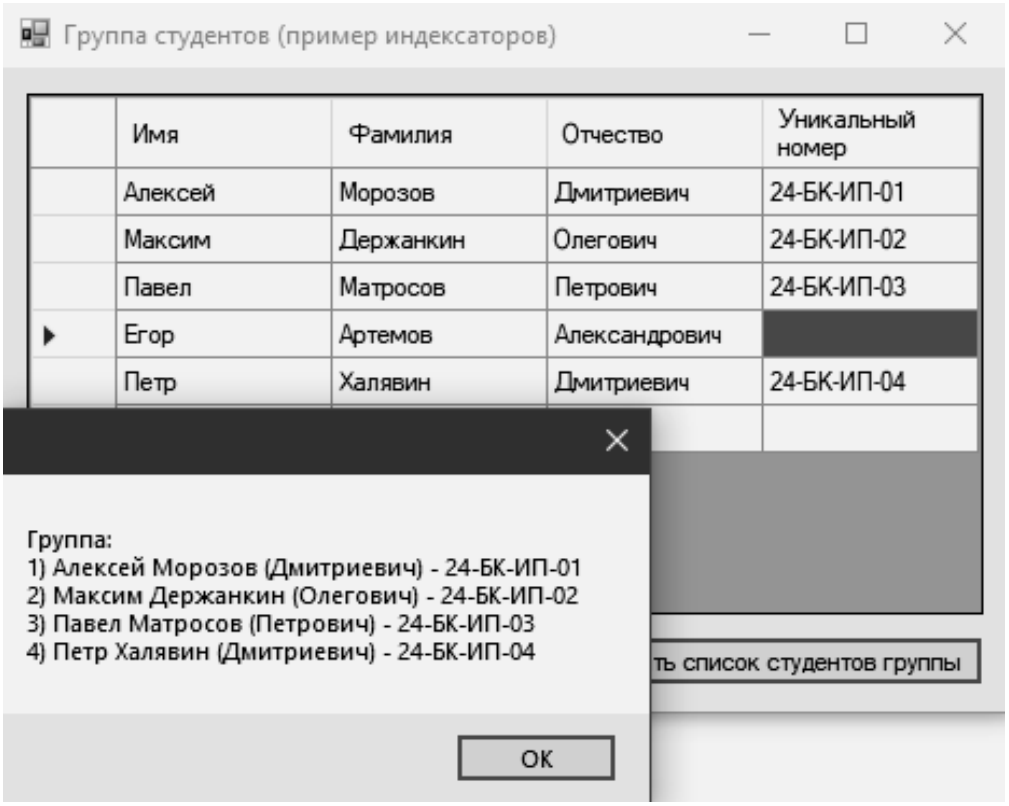

Рис. 12. Сохранены изменения, вывелись все студенты, кроме одного, где не заполнены все данные

Контрольные вопросы

1. Что называется классом?
2. Синтаксис объявления класса.
3. Перечислить элементы класса.
4. Какие спецификаторы допускаются для не вложенных классов, которые описываются непосредственно в пространстве имен?
5. Какой спецификатор доступа подразумевается для не вложенных классов по умолчанию?
6. С помощью какой операция создаётся экземпляр класса?
7. К каким типам данных относится класс?
8. Какие элементы классов присутствуют в единичном экземпляре в объекте класса?
9. Перечислите спецификаторы данных класса.
10. Какой спецификатор доступа элементов класса считается по умолчанию?
11. Имеют ли методы класса непосредственный доступ к его закрытым полям?

12. С помощью какой операции можно обратиться к полю класса?
13. Как обратиться к статическому полю класса и к константе?
14. Как обратиться к нестатическому элементу класса?
15. Для чего применяется параметр this в явном виде?
16. Для чего предназначен конструктор?
17. Перечислить свойства конструктора.
18. В каких случаях конструктор базового класса вызывается явным образом?
19. С помощью какого ключевого слова можно явным образом вызвать конструктор базового класса?
20. Для чего служит свойство класса?
21. Синтаксис объявления свойства.
22. Могут ли отсутствовать части get/set или обе одновременно?
23. Когда свойство доступно только для чтения (read-only)?
24. Когда свойство доступно только для записи (write-only)?
25. Какие спецификаторы доступа можно задавать для частей свойства, объявленного со спецификатором public?
26. Какие спецификаторы доступа можно задавать для частей свойства, объявленного со спецификатором protected internal?

2 Иерархия классов

2.1 Наследование

Управлять большим количеством разрозненных классов довольно сложно. С этой проблемой можно справиться путем упорядочивания и ранжирования классов, то есть объединяя общие для нескольких классов свойства в одном классе и используя его в качестве базового. Эту возможность предоставляет механизм наследования.

Наследование применяется для следующих взаимосвязанных целей:

- исключения из программы повторяющихся фрагментов кода;
- упрощения модификации программы;
- упрощения создания новых программ на основе существующих.

Более того, наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

Кроме механизма наследования также будут рассмотрены такие важные понятия ООП как полиморфизм и инкапсуляция, которые принимают участие в формировании иерархии классов.

Вспомним синтаксис класса:

```
[атрибуты] [спецификаторы] class имя_класса [: предок]
{
  тело_класса
}
```

При описании класса имя его предка записывается в заголовке класса после двоеточия. Класс, который наследуется, называется базовым. Класс, который наследует, называется производным. Производный класс наследует все переменные, методы, свойства, операторы и индексаторы, определенные в базовом классе, кроме того, в производный класс могут быть добавлены уникальные элементы или переопределены существующие. Если имя предка явным образом не указано, то предком считается базовый класс всех типов данных в языке C#, т.е. тип object.

Рассмотрим наследование классов на примере геометрических фигур. В качестве базового класса создадим класс PointPlane (точка на плоскости), в качестве производного класса от PointPlane класс PointSpace (точка в пространстве):

```
using System;
```

```
namespace MyProgram
{
  public class PointPlane //Базовый класс – точка на плоскости
  {
    public int x;
    public int y;
    public void Show()
    {
      Console.WriteLine("({0}, {1})", x, y);
    }
  }
}

using System;

namespace MyProgram
{
  public class PointSpace : PointPlane //Производный класс – точка в пространстве
  {
    public int z;
    public void Show()
    {
      Console.WriteLine("({0}, {1}, {2})", x, y, z);
    }
  }
}
```

Рассмотрим на примере использование созданных классов:

```
using System;

namespace MyProgram
{
  class Program
  {
    static void Main()
    {
      PointPlane pointP=new PointPlane();
      pointP.x=10;
      pointP.y=10;
      pointP.Show();
```

```
        PointSpace pointS=new PointSpace();
        pointS.x=1;
        pointS.y=2;
        pointS.z=3;
        pointS.Show();
    }
  }
}
```

Результат работы программы:
(10, 10)
(1, 2, 3)

Экземпляр класса PointSpace с одинаковой легкостью использует как собственные поля, так и унаследованные от класса PointPlane. Но в двух классах определен метод Show, поэтому компилятором будет сгенерировано предупреждение.

Чтобы избежать подобного предупреждения необходимо перед одноименным членом производного класса (в данном случае перед методом Show в классе PointSpace) поставить спецификатор new. Данный спецификатор скрывает одноименный член базового класса от производного и позволяет полностью его переопределить.

Использование защищенного доступа

В данном примере поля x и y базового класса были открыты для доступа (public). Если убрать public, то поля автоматически станут закрытыми для доступа (private), в том числе и для доступа из производного класса. Решить проблему доступа к закрытым полям базового класса из производного можно двумя способами: используя свойства класса или спецификатор protected. При объявлении какого-то члена класса с помощью спецификатора protected, он становится закрытым для всех классов, кроме производных.

Наследование конструкторов

В иерархии классов как базовые, так и производные классы могут иметь собственные конструкторы. При этом конструктор базового класса создает часть объекта, соответствующую базовому классу, а конструктор производного класса – часть объекта, соответствующую производному классу. Так как базовый класс не имеет доступа к элементам производного класса, то их конструкторы должны быть раздельными.

В предыдущем примере классы создавались за счет автоматического вызова средствами C# конструктора по умолчанию. Добавим конструктор только в производный класс PointSpace.

```
using System;

namespace MyProgram
{
   public class PointPlane //Базовый класс
  {
     //поля доступны только из производных классов
     protected int x;
     protected int y;
     public void Show()
    {
       Console.WriteLine("({0}, {1})", x, y);
    }
  }
}

using System;

namespace MyProgram
{
   public class PointSpace:PointPlane
  {
     protected int z; //поле доступно только из производных классов
     public PointSpace(int x, int y, int z) //конструктор производного класса
    {
       this.x=x;
       this.y=y;
       this.z=z;
    }
    public new void Show()
   {
        Console.WriteLine("({0}, {1}, {2})", x, y, z);
    }
  }
}
```

В данном случае конструктор определяется только в производном классе, поэтому часть объекта, соответствующая базовому классу, создается автоматически с помощью конструктора по умолчанию, а часть объекта, соответствующая производному классу, создается собственным конструктором.

Обратите внимание на то, что для производного класса возможно использовать параметр base, который действует подобно параметру this, за исключением того, что base всегда ссылается на базовый класс. В данном контексте параметр base позволяет получить доступ к члену базового класса, который скрыт за членом производного класса. Формат использования параметра:

base.член_класса

В качестве *член_класса* можно указывать либо метод, либо поле экземпляра.

Рассмотрим использование модифицированных классов на следующем примере:

```
PointPlane pointP = new PointPlane();
pointP.Show();
PointSpace pointS = new PointSpace(1, 2, 3);
pointS.Show();
```

Результат работы фрагмента программы:
(0, 0)
(1, 2, 3)

Теперь добавим конструктор в базовый класс.

```
using System;

namespace MyProgram
{
  public class PointPlane
 {
    protected int x;
    protected int y;
    public PointPlane(int x, int y) //конструктор базового класса
   {
       this.x=x;
       this.y=y;
   }
   public void Show()
  {
     Console.WriteLine("({0}, {1})", x, y);
  }
```

```
    }
}
```

Если теперь мы попытаемся скомпилировать программу, то получим сообщение об ошибке. Дело в том, что в .NET объявление в классе конструктора с параметрами автоматически приводит к удалению конструктора по умолчанию, если он не был явно задан разработчиком. При этом конструктор класса-потомка автоматически пытается обратиться к конструктору по умолчанию, которого теперь больше нет.

Для того, чтобы решить эту проблему, необходимо явно указать в конструкторе класса-потомка, какой конструктор класса-родителя необходимо вызвать. Это делается с помощью служебного слова base следующим образом:

```
public PointSpace(int x, int y, int z):base (x, y) //конструктор производного класса
{
   this.z=z;
}
```

В данном случае параметр base перед выполнением конструктора производного класса вызывает конструктор базового класса, передавая ему в качестве параметров список переменных.

Рассмотрим использование модифицированных классов на примере:

```
PointPlane pointP = new PointPlane(10, 10);
pointP.Show();
PointSpace pointS = new PointSpace(1, 2, 3);
pointS.Show();
```

Результат работы фрагмента программы:

```
(10, 10)
(1, 2, 3)
```

В общем случае с помощью параметра base можно вызвать конструктор любой формы, определенный в базовом классе. Реально же выполнится тот конструктор, список формальных параметров которого будет соответствовать списку аргументов, переданных base.

Рассмотрим модификацию классов PointPlane и PointSpace, содержащих по несколько конструкторов:

```
using System;
```

```
namespace MyProgram
{
  public class PointPlane
 {
    protected int x;
    protected int y;
    public PointPlane()
   { }

    public PointPlane(int a)
   {
     x=a;
     y=a;
   }

    public PointPlane(int x, int y)
   {
     this.x=x;
     this.y=y;
   }

    public void Show()
   {
     Console.WriteLine("({0}, {1})", x, y);
   }
  }
}

using System;

namespace MyProgram
{
  public class PointSpace:PointPlane
 {
    protected int z;
    public PointSpace()
   { }

    public PointSpace(int a) :base (a)
   {
      z=a;
   }
```

```
    public PointSpace(int x, int y, int z) :base (x, y)
    {
      this.z=z;
    }

    public new void Show()
    {
      Console.WriteLine("({0}, {1}, {2})", x, y, z);
    }
  }
}
```

Инициируем вызов различных конструкторов и посмотрим, что из этого получится:

```
PointPlane pointP1 = new PointPlane();
pointP1.Show();
PointPlane pointP2 = new PointPlane(1);
pointP2.Show();
PointPlane pointP3 = new PointPlane(2, 3);
pointP3.Show();
PointSpace pointS1 = new PointSpace();
pointS1.Show();
PointSpace pointS2 = new PointSpace(4);
pointS2.Show();
PointSpace pointS3 = new PointSpace(5, 6, 7);
pointS3.Show();
```

Результат работы фрагмента программы:

```
(0, 0)
(1, 1)
(2, 3)
(0, 0, 0)
(4, 4, 4)
(5, 6, 7)
```

2.3 Класс object

Все C#-типы, включая размерные типы, выведены из класса object. Следовательно, ссылку типа object можно использовать в качестве ссылки на любой другой тип, в том числе размерный.

Если ссылка типа object указывает на значение нессылочного типа, то происходит приведение к объектному типу (boxing). В результате этого процесса значение нессылочного типа должно сохраниться подобно объекту или экземпляру класса. Другими словами, «необъектное» значение помещается в объектную оболочку и размещается в динамической памяти. Такой «необъектный» объект можно затем использовать подобно любому другому объекту. Приведение к объектному типу происходит автоматически.

Восстановление значения из «объектного образа» называется unboxing. Это действие выполняется с помощью операции приведения типа, т.е. приведения ссылки на объект класса object к значению желаемого типа.

Рассмотрим простой пример, который иллюстрирует приведение значения к объектному типу и его восстановление.

```
int x=1;
object ob=x; //boxing
int y=(int) ob; //unboxing
Console.WriteLine("y = {0}", y);
```

Результат работы фрагмента программы:

```
y = 1
```

Так как C# является языком со строгой типизацией, в нем требуется строгое соблюдение совместимости типов с учетом стандартных преобразований типов. Поэтому ссылка одного типа обычно не может ссылаться на объект другого ссылочного типа за одним небольшим исключением – ссылочная переменная базового класса может ссылаться на объект любого производного класса.

В нашем случае допустимой будет следующая команда:

```
PointPlane pointS = new PointSpace(1,2,3);
```

Более того, т.к. тип object является базовым для всех типов данных, становится допустимой следующая последовательность команд:

```
object ob1 = new PointPlane(4, 5);
object ob2 = new PointSpace(6, 7, 8);
```

Ошибка возникнет при попытке обратиться к методу Show. Например, команда

```
pointS.Show();
```

вместо ожидаемого (1, 2, 3) выведет (1, 2).

А команда:

ob1.Show();

вообще не сможет выполниться, т.к. для класса object метод Show не определен.

Давайте разберемся, почему вместо (1, 2, 3) мы получили (1, 2). Как уже упоминалось ранее, все методы класса находятся в памяти в единственном экземпляре и используются всеми объектами одного класса совместно. Это решение вполне логично, так как если у нас есть 1000 объектов типа PointPlane, то 1000 раз дублировать код метода Show нет никакого смысла.

В результате в случае с присваиванием ссылке на класс PointPlane объекта PointSpace возникает конфликт между двумя таблицами методов – базового класса и класса-потомка.

Так как потомков у базового класса может быть много (в том числе потомков третьего и более высоких уровней) и поиск по всем возможным таблицам методов является достаточно длительным процессом, компилятор считает, что если не сказано обратное, то какой метод будет вызываться определяется типом ссылки на объект. Для того чтобы явно указать на необходимость поиска нужного метода по типу объекта, а не ссылки, используются спецификаторы virtual и abstract, а обозначенные ими методы принято называть виртуальными и абстрактными соответственно.

```
public class Object
{
    public Object();

    ~Object();

    public static bool Equals(Object objA, Object objB);
    public static bool ReferenceEquals(Object objA, Object objB);
    public virtual bool Equals(Object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
    protected Object MemberwiseClone();
}
```

Рис. 13. Структура класса Object

2.4 Виртуальные методы

Виртуальный метод – это метод, который объявлен в базовом классе с использованием ключевого слова virtual, и затем переопределен в производном классе с помощью ключевого слова override. При этом если реализовано наследование, в том числе и многоуровневое, то каждый производный класс может иметь свою собственную версию виртуального метода.

Этот факт особенно полезен в случае, когда доступ к объекту производного класса осуществляется через ссылочную переменную базового класса. В этой ситуации CLR сама выбирает, какую версию виртуального метода нужно вызвать. Этот выбор производится по типу объекта, на который ссылается данная ссылка.

Модифицируем методы Show в классах PointPlane и PointSpace с учетом сказанного.

```
public virtual void Show() //в классе PointPlane
{
   Console.WriteLine("({0}, {1})", x, y);
}

public override void Show() //в классе PointSpace
{
   Console.WriteLine("({0}, {1}, {2})", x, y, z);
}
```

Теперь рассмотрим следующий фрагмент программы:

```
PointPlane []array=new PointPlane [6];
array[0]=new PointPlane();
array[1]=new PointPlane(1);
array[2]=new PointPlane(2, 3);
array[3]=new PointSpace();
array[4]=new PointSpace(4);
array[5]=new PointSpace(5, 6, 7);
foreach (PointPlane item in array)
{
   item.Show();
}
```

Результат работы фрагмента программы:

(0, 0)
(1, 1)
(2, 3)
(0, 0, 0)
(4, 4, 4)
(5, 6, 7)

Таким образом, благодаря полиморфизму через ссылочную переменную базового класса можно обращаться к объектам разного типа, а также с помощью одного и того же имени выполнять различные действия.

2.5 Абстрактные классы

Иногда полезно создать базовый класс, определяющий только своего рода «пустой бланк», который унаследуют все производные классы, причем каждый из них заполнит этот «бланк» собственной информацией. Такой класс определяет структуру методов, которые производные классы должны реализовать, но сам класс при этом не обеспечивает реализации этих методов. Подобная ситуация может возникнуть тогда, когда базовый класс попросту не в состоянии реализовать метод. В данной ситуации разрабатываются абстрактные методы или целые абстрактные классы.

Абстрактный метод описывается с помощью спецификатора abstract. Он не имеет тела и, следовательно, не реализуется базовым классом, а производные классы должны его обязательно переопределить. Абстрактный метод автоматически является виртуальным и использовать спецификатор virtual не нужно. Более того, если вы попытаетесь использовать два спецификатора одновременно, abstract и virtual, то компилятор выдаст сообщение об ошибке.

Если класс содержит один или несколько абстрактных методов, то его также нужно объявить как абстрактный, используя спецификатор abstract перед class. Поскольку абстрактный класс полностью не реализован, то невозможно создать экземпляр класса с помощью операции new. Например, если класс Demo определен как абстрактный, то попытка создать экземпляр класса Demo повлечет ошибку:

```
Demo a = new Demo();
```

Однако можно создать массив ссылок, тип которых соответствует абстрактному классу:

```
Demo[] Ob = new Demo[5];
```

Если производный класс наследует абстрактный, то он должен полностью переопределить все абстрактные методы базового класса или также быть объявлен как абстрактный. Таким образом, спецификатор abstract наследуется до тех пор, пока в производном классе не будут реализованы все абстрактные методы.

Рассмотрим пример использования абстрактных методов и классов.

```
using System;

namespace MyProgram
{
   abstract public class Point //абстрактный класс
  {
     abstract public void Show();
     abstract public double Distance();
  }
}
```

Класс Point содержит объявление двух абстрактных методов, которые при наследовании необходимо будет реализовать. В общем случае абстрактный класс может содержать не только абстрактные методы.

Используя абстрактный класс, модифицируем классы PointPlane и PointSpace.

```
using System;
namespace MyProgram
{
   public class PointPlane: Point
  {
      protected int x;
      protected int y;

      public PointPlane()
     { }

      public PointPlane(int a)
     {
        x=a;
        y=a;
     }
     public PointPlane(int x, int y)
```

```
        {
            this.x=x;
            this.y=y;
        }

        public override void Show() //переопределяем абстрактный метод
        {
            Console.WriteLine("({0}, {1})", x, y);
        }

        public override double Distance() //переопределяем абстрактный
метод
        {
            return Math.Sqrt(x*x+y*y);
        }
    }
}

using System;

namespace MyProgram
{
    public class PointSpace:PointPlane
    {
        protected int z;
        public PointSpace()
        { }

        public PointSpace(int a):base (a)
        {
            z=a;
        }

        public PointSpace(int x, int y, int z):base (x, y)
        {
            this.z=z;
        }

        public override void Show() //переопределяем абстрактный метод
        {
            Console.WriteLine("({0}, {1}, {2})", x, y, z);
```

```
        }

        public override double Distance() //переопределяем абстрактный
метод
        {
            return Math.Sqrt(x*x+y*y+z*z);
        }
      }
    }
```

Теперь рассмотрим следующий фрагмент программы:

```
Point[] array = new Point[6];
array[0] = new PointPlane();
array[1] = new PointPlane(1);
array[2] = new PointPlane(2, 3);
array[3] = new PointSpace();
array[4] = new PointSpace(4);
array[5] = new PointSpace(5, 6, 7);
foreach (Point item in array)
{
    item.Show();
    Console.WriteLine("Расстояние до начала координат: {0:f2}",
item.Distance());
    Console.WriteLine();
}
```

Результат работы фрагмента программы:

```
(0, 0)
Расстояние до начала координат: 0.00
(1, 1)
Расстояние до начала координат: 1.41
(2, 3)
Расстояние до начала координат: 3.61
(0, 0, 0)
Расстояние до начала координат: 0.00
(4, 4, 4)
Расстояние до начала координат: 6.93
(5, 6, 7)
Расстояние до начала координат: 10.49
```

2.6 Бесплодные классы

В C# при описании класса может использоваться спецификатор sealed, который запрещает производить наследование от данного класса. Например:

```
sealed class Demo { … }
class newDemo: Demo { … } // ошибка
```

2.7 Многоуровневая иерархия

До сих пор мы рассматривали простой тип иерархии классов, который состоит из одного базового и одного производного класса. В общем случае можно построить иерархию классов, состоящую из любого количества уровней наследования.

Рассмотрим структуру следующей иерархии классов:

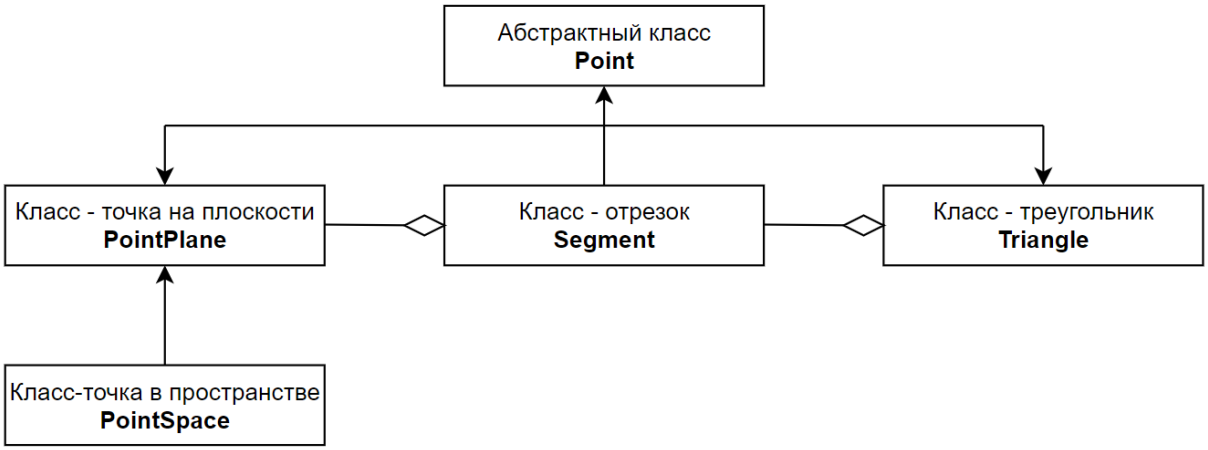

Рис. 14. Иерархия классов

Стрелки с окончанием в виде черного треугольника показывают направление наследования, а стрелки с белым ромбом на конце – агрегацию, т.е., говорят о том, что поля класса Segment будут являться ссылками на объекты класса PointPlane, а поля класса Triangle, в свою очередь, будут являться ссылками на объекты класса Segment.

Рассмотрим реализацию каждого класса.

Класс Point:

```
using System;
namespace MyProgram
```

```
{
 abstract public class Point
 {
   abstract public string Name(); //возвращает имя объекта
   abstract public void Show(); //выводит объект на экран
   abstract public double Distance();//рассчитывает длину
   abstract public void Set(params int[] ar);//устанавливает поля объекта
 }
}
```

Класс PointPlane:

```
using System;

namespace MyProgram
{
 public class PointPlane: Point
 {
   protected int x;
   protected int y;
   readonly string name="point";
   public PointPlane(int x, int y)
   {
     this.x=x;
     this.y=y;
   }

   public override string Name ()
   {
     return name;
   }

   public override void Show()
   {
     Console.WriteLine("({0}, {1})", x, y);
   }
   public override double Distance()
   {
     return Math.Sqrt(x*x+y*y);
   }
```

```
    public override void Set(params int [] ar)
    {
      if (ar.Length==2)
      {
          this.x=ar[0];
          this.y=ar[1];
       }
      else
      {
          Console.WriteLine("Неверное количество аргументов");
      }
    }

    public int X
    {
      get
      {
         return x;
      }
      set
      {
          x=value;
      }
    }

     public int Y
    {
     get
    {
       return y;
    }
     set
    {
       y=value;
     }
   }
  }
}
```

Класс PoinSpace:

```
using System;
```

```
namespace MyProgram
{
  public class PointSpace:PointPlane
  {
     protected int z;
     public PointSpace(int x, int y, int z) :base (x, y)
     {
       this.z=z;
     }

     public override void Show()
     {
       Console.WriteLine("({0}, {1}, {2})", x, y, z);
     }

     public override double Distance()
     {
       return Math.Sqrt(x*x+y*y+z*z);
     }

     public override void Set(params int [] ar)
     {
       if (ar.Length==3)
       {
          this.x=ar[0];
          this.y=ar[1];
          this.z=ar[2];
       }
       else
       {
         Console.WriteLine("Неверное количество аргументов");
       }
  }

  public int Z
 {
   get
  {
    return z;
  }
   set
```

```
        {
          z=value;
        }
       }
      }
     }
```

Класс Segment:

```
using System;

namespace MyProgram
{
  class Segment : Point
  {
    internal PointPlane begin;
    internal PointPlane end;
    readonly string name="segment";

    public Segment(int x1, int y1, int x2, int y2)
    {
      begin = new PointPlane(x1,y1);
      end = new PointPlane(x2, y2);
    }

    public override string Name ()
    {
      return name;
    }

   public override void Show ()
   {
     Console.Write("Начало отрезка");
     begin.Show();
     Console.Write("Конец отрезка");
     end.Show();
   }
   public override double Distance ()
   {
     return Math.Sqrt(Math.Pow(begin.X-end.X,2) + Math.Pow(begin.Y-
end.Y, 2));
```

```
    }

    public override void Set(params int [] ar)
    {
      if (ar.Length == 4)
      {
          begin.Set(ar[0],ar[1]);
          end.Set(ar[2],ar[3]);
      }
      else
      {
         Console.WriteLine("Неверное количество аргументов");
      }
    }

   public PointPlane Begin
    {
      get
       {
          return begin;
       }
      set
      {
         begin = value;
      }
      }

     public PointPlane End
    {
      get
      {
        return end;
      }
      set
      {
        end = value;
      }
    }
  }
}
```

Класс Triangle:

```
using System;

namespace MyProgram
{
   public class Triangle:Point
  {
     internal Segment sideA;
     internal Segment sideB;
     internal Segment sideC;
     readonly string name = "triangle";

     public Triangle(int x1, int y1, int x2, int y2, int x3, int y3)
    {
        sideA = new Segment(x1, y1, x2, y2);
        sideB = new Segment(x2, y2, x3, y3);
        sideC = new Segment(x3, y3, x1, y1);
    }

    public override string Name()
   {
        return name;
    }

    public override void Show()
   {
        Console.Write("Первая вершина");
        sideA.Begin.Show();
        Console.Write("Вторая вершина");
        sideB.Begin.Show();
        Console.Write("Третья вершина");
        sideC.Begin.Show();
  }

   public override double Distance()
  {
     return sideA.Distance() + sideB.Distance() + sideC.Distance();
  }

  public override void Set(params int[] ar)
```

```
        {
          if (ar.Length == 6)
          {
             sideA.Set(ar[0], ar[1], ar[2], ar[3]);
             sideB.Set(ar[2], ar[3], ar[4], ar[5]);
             sideC.Set(ar[4], ar[5], ar[0], ar[1]);
          }
          else Console.WriteLine("Неверное количество аргументов");
        }
      }
    }
```

Контрольные вопросы

1. Дать определение наследованию.
2. Сколько предков может иметь класс в C#?
3. Сколько потомков может иметь класс в C#?
4. Синтаксис объявителя производного класса.
5. Наследуются ли конструкторы?
6. Наследуются ли поля, методы, свойства?
7. С помощью какого ключевого слова можно переопределить элемент базового класса?
8. Метод производного класса замещает метод базового класса. Как обратится к нему из метода производного класса?
9. Какой процесс называется ранним связыванием?
10. Какой процесс называется поздним связыванием?
11. Какие методы реализуют в C# процесс позднего связывании?
12. Синтаксис объявителя виртуальных методов.
13. С помощью какого ключевого слова переопределяется виртуальный метод в производном классе?
14. Синтаксис объявителя переопределенного виртуально метода в производном классе.
15. Какие ограничения накладываются на переопределённый виртуальный метод?
16. Нужно ли переопределять виртуальные методы в каждом производственном классе?
17. Какие классы называются абстрактными?
18. Может абстрактный класс содержать полностью определённые методы?

19. С помощью какого ключевого слова объявляется бесплодные классы?

20. Перечислить открытые методы класса System.Object.

21. Переопределяются ли открытые методы класса System.Object в производственном классе?

3 Интерфейсы

3.1 Синтаксис интерфейса

Интерфейс представляет собой чисто логическую конструкцию, описывающую функциональные возможности без конкретной их реализации. С точки зрения синтаксиса интерфейсы подобны абстрактным классам. Но в интерфейсе ни у одного из методов не должно быть тела. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс – множество методов.

Отличие интерфейса от абстрактного класса:

– элементы интерфейса по молчанию имеют спецификатор доступа public и не могут иметь спецификаторов, заданных явным образом;

– интерфейс не может содержать полей и обычных методов – все элементы интерфейса должны быть абстрактными;

– класс, в списке предков которого задается интерфейс, должен переопределять все его элементы.

Членами интерфейса могут быть методы, свойства, индексаторы и события. Павловская приводит следующий синтаксис интерфейса:

```
interface имя
{
  возвращаемый_тип имя_метода1(список_параметров);
  возвращаемый_тип имя_метода2(список_параметров);
  // ...
  возвращаемый_тип имя_методаN(список_параметров);
}
```

Пример:

```
interface IPerson
{
   string Name { get; set; }
   int Age { get; set; }
   DateTime BirthDay { get; set; }
   void GetInformation();
}
```

Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы.

Кроме того, ни один из членов интерфейса не может быть объявлен как static.

Все методы интерфейса должны быть реализованы в каждом классе, включающем в себя этот интерфейс. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. В одном классе может быть реализовано любое количество интерфейсов.

3.2 Реализация интерфейса

В списке предков класса сначала указывается его базовый класс, если он есть, а затем через запятую интерфейсы, которые реализует этот класс.

Пример:

```
class Student : IPerson
{
   string name;
   int age;
   DateTime birth;
   public Student (string name, int age, DateTime birth)
   {
      Name = name
      Age = age;
      BirthDay = birth;
   }
    public string Name
    {
        get { return name;}
        set { name = value; }
    }
    public int Age
    {
        get { return age; }
        set { age = value; }
    }
    public DateTime BirthDay
    {
        get { return birth; }
        set { birth = value; }
    }
    public void GetInformation()
```

```
        {
            MessageBox.Show("Имя: "+ Name + " Возраст: " + Age + " Дата
рождения: " + BirthDay.Date);
            MessageBox.Show("Родился в " + BirthDay.DayOfWeek);
        }
    }
```

Для реализуемых элементов интерфейса в классе следует указывать спецификатор *public*. К таким элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса.

Пример:

1-й способ:

```
Student new_student;
int age;
age = Convert.ToInt32(DateTime.Now.Year -
dateTimePicker1.Value.Year);
new_student = new Student(textBox1.Text, age,
dateTimePicker1.Value);
new_student.GetInformation();
```

2-й способ:

```
int age;
age = Convert.ToInt32(DateTime.Now.Year -
dateTimePicker1.Value.Year);
IPerson student = new Student(textBox1.Text, age,
dateTimePicker1.Value);
student.GetInformation();
```

Удобство второго способа проявляется при присваивании объектам типа IPerson ссылок на объекты различных классов, поддерживающих этот интерфейс.

Также существует другой способ реализации интерфейса в классе: явное указание имени интерфейса перед реализуемым элементом. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе только через объект типа интерфейс.

Пример:

```
class Student:IPerson
{ …
    void IPerson.GetInformation1()
    {
```

```
            MessageBox.Show("Имя: " + Name + " Возраст: " + Age + " Дата
рождения: " + BirthDay.Date);
            MessageBox.Show("Родился в " + BirthDay.DayOfWeek);
        }
    }
```

При явном задании имени реализуемого интерфейса соответствующий метод не входит в интерфейс класса.

Также явное задание реализуемого типа интерфейса перед именем метода позволяет избежать конфликтов при множественном наследовании, если элементы с одинаковыми именами или сигнатурой встречаются более чем в одном интерфейсе.

Результат работы программы:

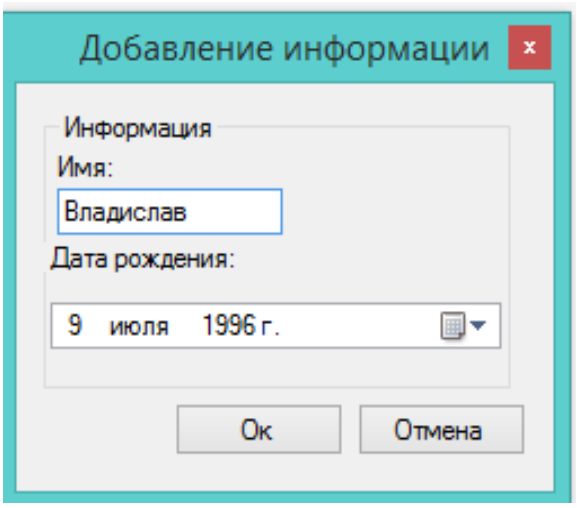

Рис. 15. Добавление информации

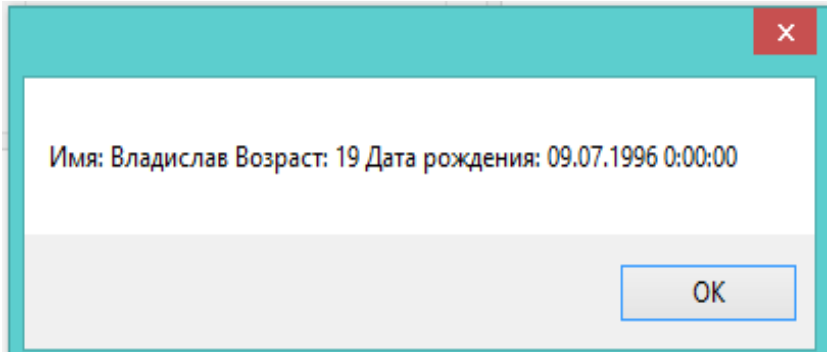

Рис. 16. Вывод данных

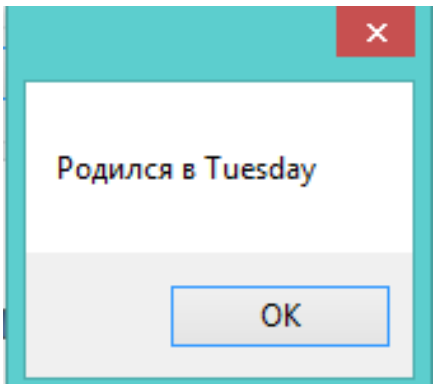

Рис. 17. Вывод данных

3.3 Работа с объектами через интерфейсы

При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции *is*. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова *is*, с типом, заданным справа.

Результат операции равен *true*, если объект можно преобразовать к заданному типу, и *false* в противном случае. Операция обычно используется в следующем контексте:

```
if (выражение is тип )
{
  // выполнить преобразование "выражения" к "типу"
  // выполнить действия с преобразованным объектом
}
```

Недостатком использования операции *is* является то, что преобразования фактически выполняется дважды: при проверке и преобразовании.

Иногда преобразование типов требуется произвести во время выполнения, но не генерировать исключение. В таком случае более эффективной будет операция *as*.

Синтаксис операции:

```
выражение as  тип
```

Если исход такого преобразования оказывается удачным, то возвращается ссылка на тип, а иначе – null.

Использование операций is и as реализовано в примере.
Пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication8
{
    class A { }
    class B : A { }
    class Program
    {
        static void Main(string[] args)
        {
            A a1 = new A();
            B b1 = new B();
            if (a1 is A)
                Console.WriteLine(" a1 имеет тип A");
            if (b1 is A)
            {
                Console.WriteLine(" b1 совместим с A, так как он
наследуется от класса A");
                A c1 = (A)b1;// преобразование переменной b1 к типу A
            }
            if (a1 is B)
            {
                Console.WriteLine(" a1 не совместим с B, так как класс B
является наследником класса A, а не наоборот");
            }
            A c2 = b1 as A;
            if (c2!=null)
                Console.WriteLine("Преобразование b1 к типу A
возможно");
            Console.ReadKey();
        }
    }
```

Результат работы программы:

```
a1 имеет тип A
b1 совместим с A, так как он наследуется от класса A
Преобразование b1 к типу A возможно
```

3.4 Наследование интерфейсов

Один интерфейс может наследовать другой. Синтаксис наследования интерфейсов такой же, как и у классов. Когда в классе реализуется один интерфейс, наследующий другой, в нем должны быть реализованы все члены, определенные в цепочке наследования интерфейсов.

Таким образом, интерфейсы могут быть организованы в иерархии. Как и в иерархии классов, в иерархии интерфейсов, когда какой-то интерфейс расширяет существующий, он наследует все абстрактные члены своего родителя (или родителей). Конечно, в отличие от классов, производные интерфейсы никогда не наследуют саму реализацию. Вместо этого они просто расширяют собственное определение за счет добавления дополнительных абстрактных членов. Базовые интерфейсы должны быть доступны в не меньшей степени, чем их потомки. Например, нельзя использовать интерфейс со спецификатором private или internal в качестве базового для открытого(public) интерфейса.

Пример:

```
namespace WindowsFormsApplication2
{
   public interface A { int Composition(); }

   public interface B: A { string GetInformation(); }

   class Class1 : B
   {
      int a;
      int b;
      public Class1(int a, int b)
      {
         this.a = a;
         this.b = b;
      }
```

```
            public int Composition() { return a * b; }
            public string GetInformation()
            {  string s = this.a +"x" + this.b + "=" +
Convert.ToString(Composition()) + Environment.NewLine;
                return s;
            }
        }
    }
```

Результат работы программы:

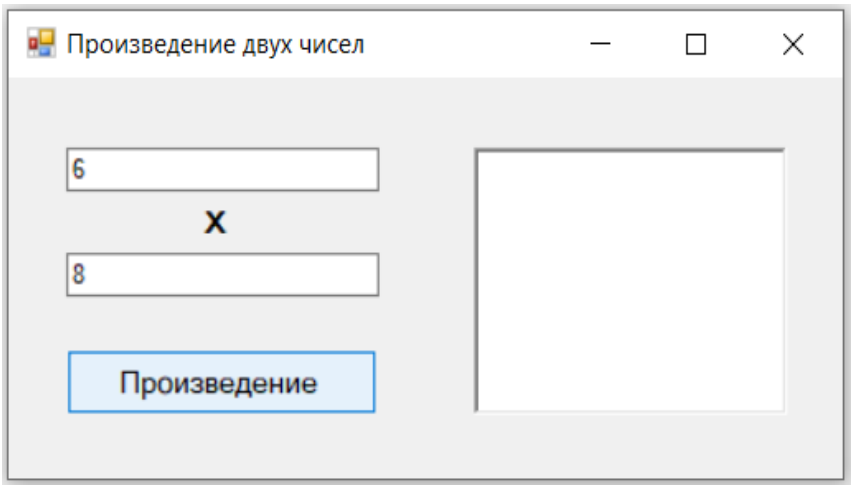

Рис. 18. Ввод данных

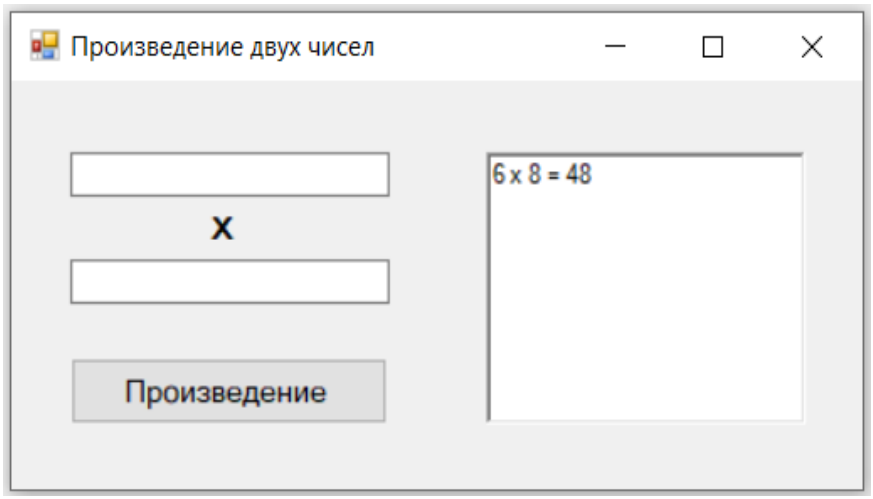

Рис. 19. Вывод результата

Обратите внимание, что класс Class1 реализует методы обоих интерфейсов A и B, иначе возникла бы ошибка при компиляции:

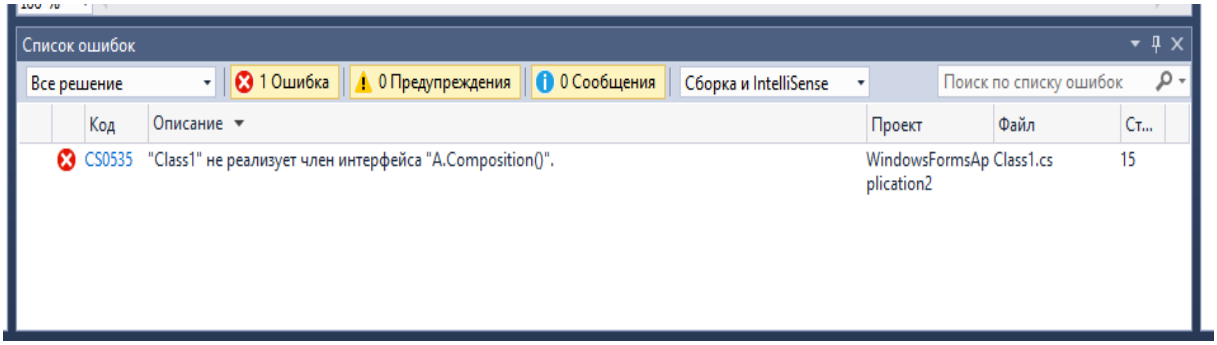

Рис. 20. Список ошибок

Корневой класс *System.Object* всей иерархии объектов .NET, называемый в C# object, обеспечивает всех наследников несколькими важными методами. Производные классы могут использовать эти методы непосредственно или переопределять их.

Открытые методы класса *System.Object*:

– Метод *Equals* с одним параметром возвращает значение *true*, если параметр и вызывающий объект ссылаются на одну и ту же область памяти.

Синтаксис:

public virtual bool Equals(object obj);

– Метод Equals с двумя параметрами возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти.

Синтаксис:

public static bool Equals(object obi, object ob2);

– Метод GetHashCode формирует хеш-код объекта и возвращает число, однозначно идентифицирующее объект. Это число используется в различных структурах и алгоритмах библиотеки. Если переопределяется метод Equals, необходимо перегрузить и метод GetHashCode.

Синтаксис:

public virtual int GetHashCode();

– Метод GetType возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвращаемое значение имеет тип Type. Это абстрактный

базовый класс иерархии, использующийся для получения информации о типах во время выполнения.

Синтаксис:

public Type Get Type() ;

– Метод ReferenceEquals возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти.

Синтаксис:

public static bool(object obi, object ob2);

– Метод ToString по умолчанию возвращает для ссылочных типов полное имя класса в виде строки, а для значимых – значение величины, преобразованное в строку. Этот метод переопределяют для того, чтобы можно было выводить информацию о состоянии объекта.

Синтаксис:

public virtual string ToString();

В примере переопределим метод *ToString*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class Sum
   {
      int z, x, y;
      public Sum(int x, int y)
      {
         this.x = x;
         this.y = y;
         z = x + y;
       }
      public override string ToString()
      {
          return z.ToString();
       }
```

```
            internal string ToString(string p)
            {
                if (p == "C") return "Денежный формат: "+z.ToString("C");
                if (p == "D5") return "Вывод целых значений с шириной поля
ввода равной 5: "+z.ToString("D5");
                if (p == "G") return "Формат общего вида: "+z.ToString("G");
                if (p == "E") return "Значение в экспоненциальном формате:
"+z.ToString("E");
                if (p == "X") return "В 16й системе счисления:
"+z.ToString("X");
                if (p == "P") return "Вывод числа в процентном формате:
"+z.ToString("P");
                if (p == "N") return "Вывод значения в формате с
разделителями разрядов: "+z.ToString("N");
                return this.ToString();
              }
        }

        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Сумма чисел 5 и 6 и их различные
выводы на экран:\n");
                Console.WriteLine("Вывод на экран с помощью
переопределенного метода ToDtring(): "+new Sum(5, 6));
                Console.WriteLine(new Sum(5, 6).ToString("E"));
                Console.WriteLine(new Sum(5, 6).ToString("C"));
                Console.WriteLine(new Sum(5, 6).ToString("D5"));
                Console.WriteLine(new Sum(5, 6).ToString("G"));
                Console.WriteLine(new Sum(5, 6).ToString("X"));
                Console.WriteLine(new Sum(5, 6).ToString("N"));
                Console.WriteLine(new Sum(5, 6).ToString("P"));
                Console.ReadKey();

            }
        }
    }
```

Результат работы программы:

```
Сумма чисел 5 и 6 и их различные выводы на экран:
Вывод на экран с помощью переопределенного метода ToDtring(): 11
  Значение в экспоненциальном формате: 1,100000E+001
  Денежный формат: 11,00 ?
  Вывод целых значений с шириной поля ввода равной 5: 00011
  Формат общего вида: 11
  В 16й системе счисления: B
  Вывод значения в формаете с разделителями разрядов: 11,00
  Вывод числа в процентном формате: 1 100,00%
```

3.5 Стандартные интерфейсы

В библиотеке классов .Net определено множество стандартных интерфейсов, задающих желаемую функциональность объектов. Например, интерфейс IComparable задает метод сравнения объектов по принципу больше и меньше, что позволяет переопределить соответствующие операции в рамках класса, наследующего интерфейс IComparable. Реализация интерфейсов IEnumerable и IEnumerator дает возможность просматривать содержимое объекта с помощью оператора foreach.

Можно создавать собственные классы, реализующие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

Интерфейс IComparable

Интерфейс IComparable определен в пространстве имен System и содержит единственный метод CompareTo, возвращающий результат сравнения двух объектов – текущего и переданного ему в качестве параметра:

```
interface IComparable
{
   int CompareTo(object obj);
}
```

Реализация данного метода должна возвращать:

1) 0 – если текущий объект и параметр равны;

2) отрицательное число, если текущий объект меньше параметра;

3) положительное число, если текущий объект больше параметра.

Использование стандартного интерфейса IComparable рассмотрим на модификации класса PointPlane.

```
public class PointPlane: Point, IComparable
{
    …
    //добавили в класс реализацию метода CompareTo

    public int CompareTo (object obj)
    {
       Point b=(Point) obj; //преобразуем параметр к типу Point
  //определяем критерии сравнения текущего объекта с параметром в
  // зависимости от удаленности точки от начала координат
      if (this.Distance()==b.Distance()) return 0;
      else if (this.Distance()>b.Distance()) return 1;
           else return -1;
    }
}
```

Обратите внимание на то, что мы изменили только класс PointPlane, класс PointSpace мы не меняли. А теперь рассмотрим следующий фрагмент программы:

```
Point [] array = new Point[5];
array[0] = new PointPlane(0, 3);
array[1] = new PointPlane(4, 3);
arrayПа = new PointSpace(0, 0, 0);
array[3] = new PointSpace(1, 1, 1);
array[4] = new PointPlane(-1, -2);
Array.Sort(array); //вызываем стандартную сортировку массива
foreach(Point item in array)
{
   item.Show(); Console.WriteLine("Расстояние до начала координат: {0:f2}", item.Distance());
  Console.WriteLine();
}
```

Результат работы фрагмента программы

```
(0, 0, 0)
Расстояние до начала координат: 0,00
(1, 1, 1)
Расстояние до начала координат: 1,73
(-1, -2)
Расстояние до начала координат: 2,24
(0, 3)
Расстояние до начала координат: 3,00
(4, 3)
Расстояние до начала координат: 5,00
```

Обратите внимание на то, что во время реализации метода CompareTo в качестве параметра передавалась ссылка на объект типа object. Напомним, что класс object является корневым классом для всех остальных в C#. Поэтому он может ссылаться на объект любого типа. Но чтобы потом получить доступ к членам объекта произвольного класса, нужно выполнить приведение типов.

Таким образом, переопределив метод CompareTo, нам удалось отсортировать массив, используя стандартный метод сортировки, определенный в классе Array.

Используя собственную реализацию метода CompareTo можно перегрузить операции отношения. Напомним, что операции отношения должны перегружаться парами: < и >, <= и >=, == и !=.

В следующем примере для класса PointPlane перегрузим операции == и != таким образом, чтобы при сравнении двух объектов возвращалось значение true, если точки находятся на равном удалении от начала координат, в противном случае – false. Для этого в класс PointPlane добавим следующие методы:

```
public static bool operator == (PointPlane a, PointPlane b)
{
  return (a.CompareTo(b)==0);
}

public static bool operator != (PointPlane a, PointPlane b)
{
  return (a.CompareTo(b)!=0);
}
```

Рассмотрим следующий фрагмент программы:

```
PointPlane a = new PointPlane(0, 3);
PointPlane b = new PointPlane(3, 0);
PointPlane c = new PointPlane(-1, -2);
if (a == b)
{
  Console.WriteLine("точки a и b равноудалены от начала координат");
}
 else
{
  Console.WriteLine("точки a и b находятся на разном расстоянии от начала координат");
}
 if (a==c)
{
  Console.WriteLine("точки a и c равноудалены от начала координат");
}
 else
{
  Console.WriteLine("точки a и c находятся на разном расстоянии от начала координат");
}
```

Результат работы фрагмента программы:
точки a и b равноудалены от начала координат
точки a и c находятся на разном расстоянии от начала координат

Однако если мы попытаемся выполнить следующий фрагмент программы:

```
Point a = new PointPlane(0, 3);
Point b = new PointPlane(3, 0, 0);
if (a == b)
{
  Console.WriteLine("точки a и b равноудалены от начала координат");
}
else
{
```

```
        Console.WriteLine("точки a и b находятся на разном расстоянии от начала координат");
    }
```

то получим следующий результат:

точки a и b находятся на разном расстоянии от начала координат

Давайте разберемся, что произошло. Дело в том, что операторы проверки на равенство были перегружены в классе PointPlane. Абстрактный класс Point про них ничего не знает и использует стандартные операторы, которые проверяют равенство ссылок, а не объектов. Например, если заменить код создания объектов на такой:

```
PointPlane a = new PointPlane(0, 3);
PointPlane b = new PointSpace(3, 0, 0);
```

то сравнение пройдет правильно, так как ссылки на PointPlane уже знают о правилах перегрузки операторов.

Решить эту проблему нам поможет создание следующей иерархии объектов:

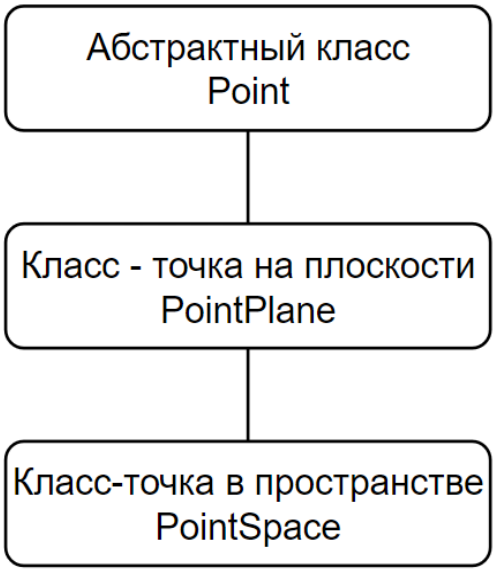

Рис. 21. Иерархия типов

Рассмотрим каждый компонент иерархии типов.

Абстрактный класс Point:

```
using System;
namespace MyProgram
```

```
{
  abstract public class Point:IComparable
 {
    abstract public string Name { get;}
    abstract public void Show();
    abstract public double Distance();
    abstract public int this [int i] { get; set;}
    public int CompareTo (object obj)
   {
       Point b=(Point) obj;
       if (this.Distance() == b.Distance())
      {
         return 0;
      }
      else if (this.Distance() > b.Distance())
           {
              return 1;
           }
           else
          {
              return -1;
          }
   }
   public static bool operator == (Point a, Point b)
  {
      return (a.CompareTo(b) == 0);
   }
   public static bool operator != (Point a, Point b)
  {
      return (a.CompareTo(b) != 0);
   }
 }
}
```

Класс PointPlane:

```
using System;
namespace MyProgram
{
   public class PointPlane: Point
  {
```

```
 protected int x;
 protected int y;
 readonly string name = "point";

 public PointPlane(int x, int y)
{
   this.x = x;
   this.y = y;
}

public override string Name
{
 get
 {
    return name;
 }
}

public override void Show()
{
   Console.WriteLine("({0}, {1})", x, y);
}

public override double Distance()
{
   return Math.Sqrt(x*x + y*y);
}

public override int this [int i]
{
 get
 {
   if (i == 1)
  {
     return x;
   }
  else
  {
     if (i == 2)
    {
       return y;
```

```
            }
             else
            {
                Console.WriteLine("Неверно указан индекс");
                return 0;
            }
        }
      }
       set
      {
        if (i == 1)
        {
            x=value;
        }
        else
        {
            if (i == 2)
            {
                y=value;
            }
            else
            {
              Console.WriteLine("Неверно указан индекс");
            }
        }
      }
    }
   }
}
```

Класс PointSpace:

```
using System;

namespace MyProgram
{
  public class PointSpace:PointPlane
  {
     protected int z;

     public PointSpace(int x, int y, int z):base (x, y)
```

```
{
   this.z = z;
}

 public new void Show()
{
   Console.WriteLine("({0}, {1}, {2})", x, y, z);
}

public new double Distance()
{
  return Math.Sqrt(x*x + y*y + z*z);
}

public new int this [int i]
{
 get
 {
   if (i == 1)
  {
      return x;
  }
  else
 {
      if (i == 2)
     {
         return y;
     }
      else
     {
        if (i == 3)
       {
          return z;
       }
       else
      {
         Console.WriteLine("Неверно указан индекс");
         return 0;
      }
    }
 }
```

```
        }
        set
        {
          if (i == 1)
          {
              x = value;
          }
           else
          {
              if (i == 2)
              {
                  y = value;
              }
               else
              {
                 if (i == 3)
                 {
                     z = value;
                 }
                  else
                 {
                      Console.WriteLine("Неверно указан индекс");
                 }
               }
            }
          }
       }
     }
```

Рассмотрим, как теперь будут сравниваться между собой разнотипные объекты:

```
Point a = new PointPlane(0, 3);
Point b = new PointSpace(3, 0, 0);
Point c = new PointSpace(3, 0, 1);
if (a == b)
 {
   Console.WriteLine("точки a и b равноудалены от начала координат");
  }
  else
```

```
        {
          Console.WriteLine("точки a и b находятся на разном расстоянии от начала координат");
        }
       if (a == c)
        {
          Console.WriteLine("точки a и c равноудалены от начала координат");
        }
       else
       {
          Console.WriteLine("точки a и c находятся на разном расстоянии от начала координат");
       }
```

Результат работы фрагмента программы:
точки a и b равноудалены от начала координат
точки a и c находятся на разном расстоянии от начала координат

Сортировка по разным критериям (интерфейс IComparer)

Интерфейс *IComparer* определен в пространстве имен *System.Collections*. Он содержит один метод *Compare*, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

```
interface IComparer
{
   int Compare (object ob1, object ob2);
}
```

Принцип применения этого интерфейса в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента (существует несколько перегруженных версий этого метода).

Пример, демонстрирующий сортировку по разным критериям с помощью IComparer:

```
using System;
```

```
using System.Collections.Generic;
using System.Windows.Forms;

namespace ExampleIComparer
{
    class Item
    {
        public string Name { get; private set; }
        public string Type { get; private set; }

        public Item(string name, string type)
        {
            Name = name;
            Type = type;
        }

        public override string ToString()
        {
            return $"{Name} ({Type})";
        }
    }

    // Сортировка предметов в инвентаре по названию
    class SortingByName : IComparer<Item>
    {
        public int Compare(Item x, Item y)
        {
            return x.Name.CompareTo(y.Name);
        }
    }

    // Сортировка предметов в инвентаре по типу
    class SortingByType : IComparer<Item>
    {
        public int Compare(Item x, Item y)
        {
            return x.Type.CompareTo(y.Type);
        }
    }

    public partial class Form1 : Form
```

```
{
    public Form1()
    {
        InitializeComponent();
        typeComboBox.SelectedIndex = 0;
        sortingTypeComboBox.SelectedIndex = 0;
    }

    // Сортировка методом вставки.
    // В аргументах передается компаратор, определяющий тип
    // сортировки (по названию, либо по типу предмета)
    private void SortInventory(IComparer<Item> sortComparer)
    {
        int itemsCount = itemListBox.Items.Count;
        for (int i = 0; i < itemsCount; i++)
        {
            int max = i;

            for (int j = i + 1; j < itemsCount; j++)
            {
                Item item = (Item)itemListBox.Items[j];
                Item itemMax = (Item)itemListBox.Items[max];
                if (sortComparer.Compare(item, itemMax) < 0)
                    max = j;
            }

            Item temp = (Item)itemListBox.Items[i];
            itemListBox.Items[i] = itemListBox.Items[max];
            itemListBox.Items[max] = temp;
        }
    }

    // По нажатию - добавление нового предмета
    private void addButton_Click(object sender, EventArgs e)
    {
        string name = nameTextBox.Text;
        string type = typeComboBox.SelectedItem.ToString();

        Item newItem = new Item(name, type);
        itemListBox.Items.Add(newItem);
    }
```

```
        // По нажатию - сортировка предметов в инвентаре
        private void sortButton_Click(object sender, EventArgs e)
        {
            string sortType = sortingTypeComboBox.SelectedItem.ToString();

            IComparer<Item> sortCompare;
            if (sortType == "По названию")
                sortCompare = new SortingByName();
            else
                sortCompare = new SortingByType();

            SortInventory(sortCompare);
        }
    }

    …
}
```

Результаты работы программы:

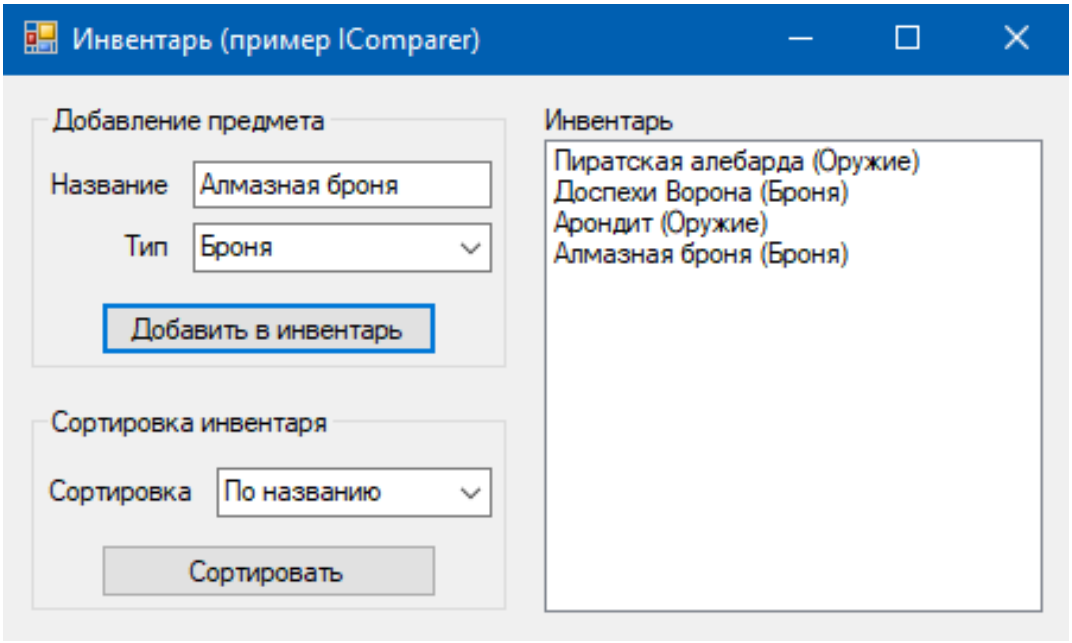

Рис. 22. Добавление предметов в инвентарь

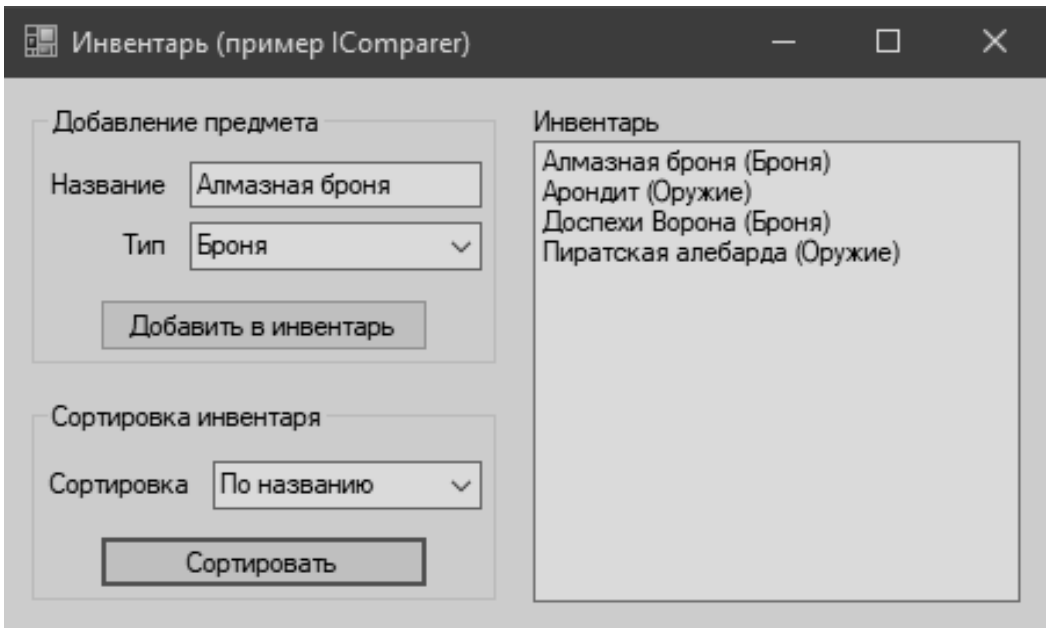

Рис. 23. Сортировка предметов по названию

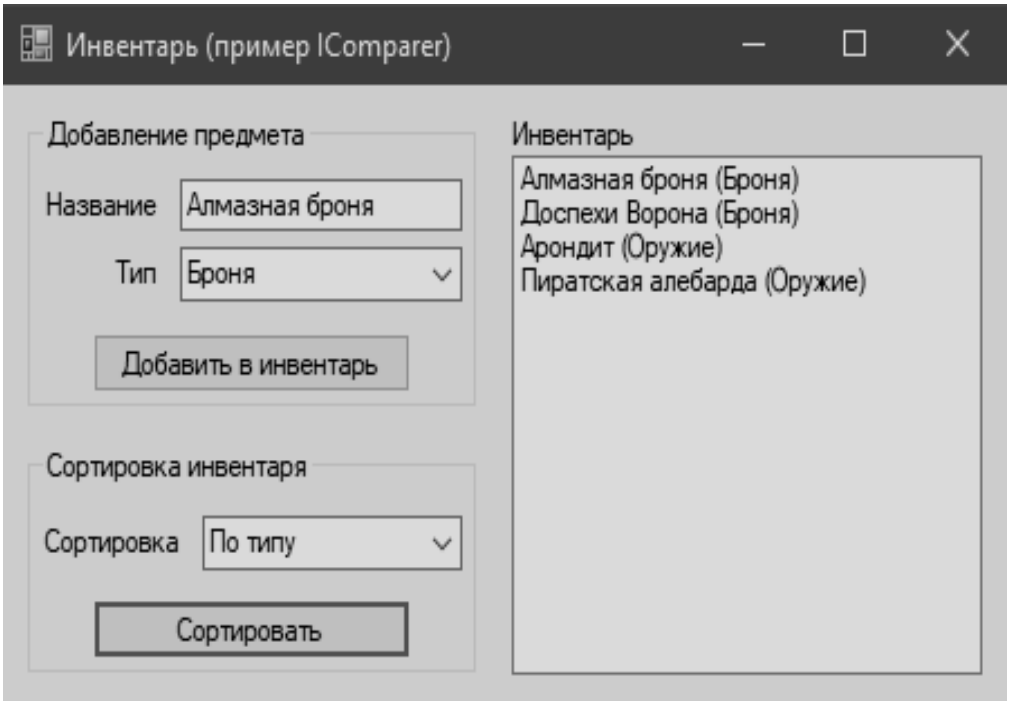

Рис. 24. Сортировка предметов по типу

Клонирование объектов (Интерфейс ICloneable)

Клонирование – это создание копии объекта. Копия объекта называется клоном. При присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект.

Для создания полностью независимых объектов необходимо глубокое клонирование.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса Icloneable и переопределять его единственный метод Clone.

В примере представлена реализация стандартных интерфейсов Icomparer, Icomparable, Icloneable.

Пример:

```
namespace Интерфейсы
{
   interface ILibrary
   {
         string Name { set; get; }
         string Family { set; get; }
         string Address { set; get; }
         string Phone { set; get; }
   }
class Library : ILibrary, IComparable<Library>, ICloneable
{
   string name;
   string family;
   public int[] birthday = new int[3];
   string address;
   string phone;
   public string Name
   {
      set { name = value; }
      get { return name; }
   }
   public string Family
   {
      set { family = value; }
      get { return family; }
   }
   public string Address
   {
      set { address = value; }
      get { return address; }
   }
```

```
        public string Phone
        {
            set { phone = value; }
            get { return phone; }
        }
        public Library(string name, string family, int d, int m, int y, string
address, string phone)
        {
            Name = name;
            Family = family;
            this.birthday[0] = d;
            this.birthday[1] = m;
            this.birthday[2] = y;
            Address = address;
            Phone = phone;
        }
        public override string ToString()
        {
            return string.Format("ФИО: {0} {1} {2}Дата рождения:
{3}.{4}.{5}{6}Город: {7} {8}Телефон: {9}{10}", Name, Family,
Environment.NewLine, birthday[0], birthday[1], birthday[2],
Environment.NewLine, Address, Environment.NewLine, Phone,
Environment.NewLine);
        }
        public int CompareTo(Library other)
        {
            if (this.birthday[2] > other.birthday[2]) return 1;
            if (this.birthday[2] < other.birthday[2]) return -1;
            return 0;
        }
        public object Clone()
        {
            return new Library(this.name, this.family, this.birthday[0],
this.birthday[1], this.birthday[2], address, phone);
        }
    }

    class Sort : IComparer<Library>
        {
            public int Compare(Library x, Library y)
            {
```

```
            if (x.birthday[2] > x.birthday[2]) return 1;
            if (x.birthday[2] < x.birthday[2]) return -1;
            return 0;
        }
    }
}
```

Результат работы программы:

Рис. 25. Результат выполнения

Проверка на равенство (интерфейс IEquatable)

Как и IComparable, интерфейс IEquatable определен в пространстве имен System и предназначен для сравнения двух объектов, однако в данном случае выполняется только проверка на равенство. IEquatable содержит один метод Equals, возвращающий значение типа bool (true – если объекты равны, false – если не равны):

```
interface IEquatable<T>
{
    bool Equals (T other);
}
```

Данный метод Equals, в отличие от System.Object.Equals, позволяет проверять равенство объектов не по ссылке, а по определенным полям класса.

Пример реализации IEquatable:

```
class Book : IEquatable<Book>
{
   string title;
   string author;
   int year;

   public Book(string title, string author, int year)
   {
      this.title = title;
      this.year = year;
      this.author = author;
   }

   public bool Equals(Book other)
   {
      // Книги равны, если равны поля title, author и year
      return title == other.title && author == other.author && year == other.year;
   }

   public override string ToString()
   {
      return $"{author} - {title} ({year} г.)";
   }
}

public partial class Form1 : Form
{
   public Form1()
   {
      InitializeComponent();
   }

   private void takeBookButton_Click(object sender, EventArgs e)
   {
```

```
            string title = titleTextBox.Text;
            string author = authorTextBox.Text;
            int year = (int)yearUpDown.Value;

            Book book = new Book(title, author, year);

            // Проверка, выбирал ли уже пользователь эту книгу
            foreach (Book takenBook in takenBooksListBox.Items)
            {
                if (book.Equals(takenBook))
                {
                    MessageBox.Show("Вы уже выбирали данную книгу.", "Ошибка");
                    return;
                }
            }

            takenBooksListBox.Items.Add(book);
        }
    }
```

Результаты работы программы можно увидеть на рисунках ниже.

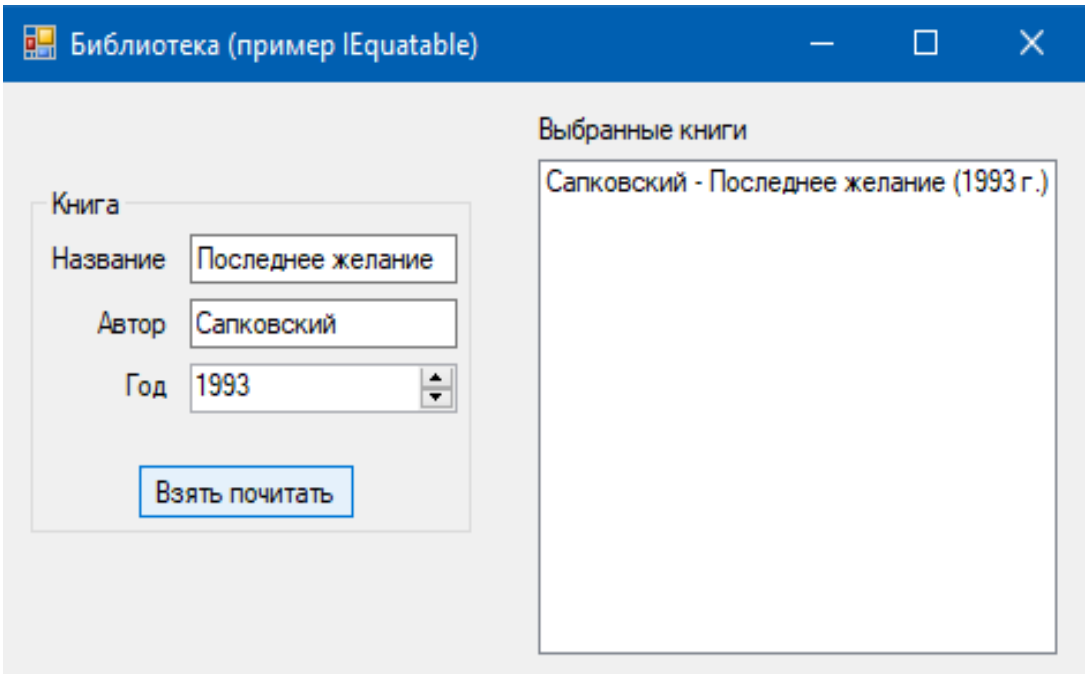

Рис. 26. Выбор первой книги для прочтения

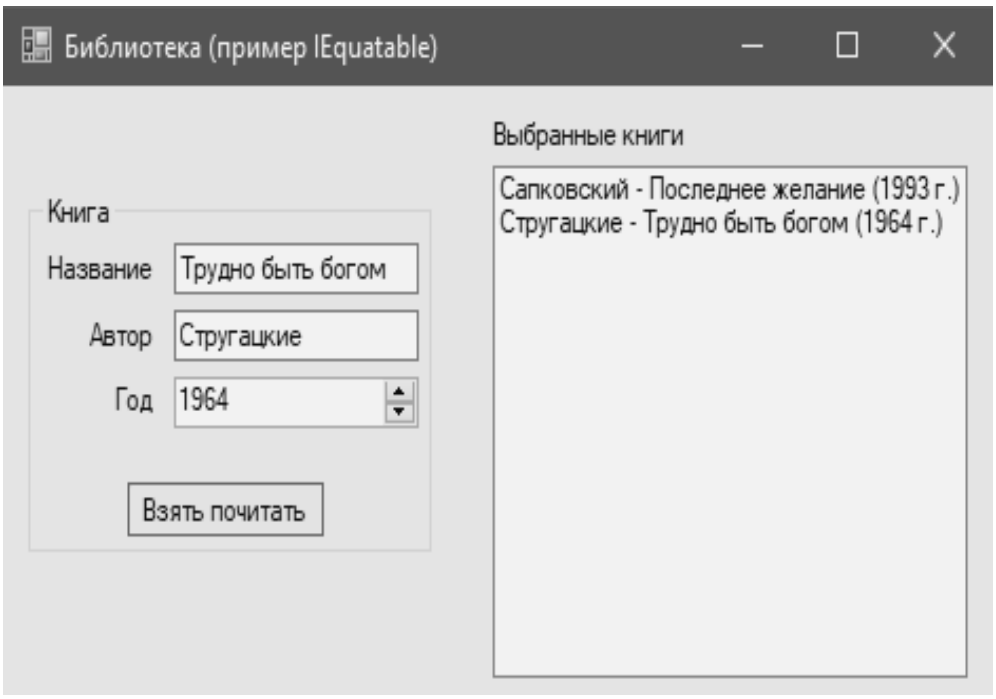

Рис. 27. Выбор второй книги для прочтения

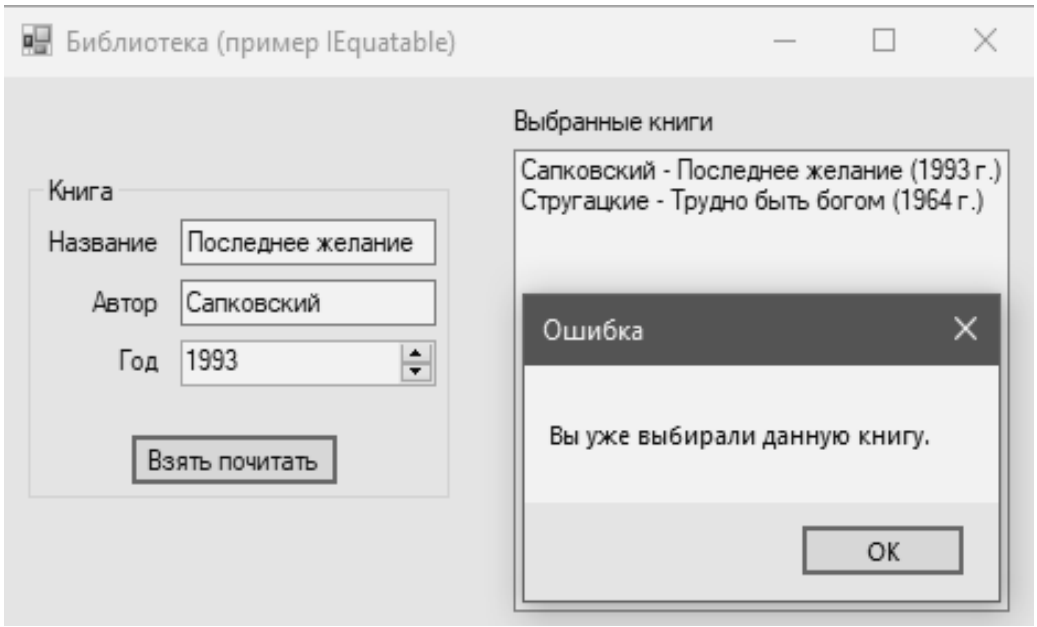

Рис. 28. Ошибка при повторном выборе первой книги

Ниже представлен еще один пример реализации IEquatable.

```
class Song : IEquatable<Song>
{
   readonly string title;
```

```
        readonly string author;

        public Song(string title, string author)
        {
            this.title = title;
            this.author = author;
        }

        public bool Equals(Song other)
        {
            // Песни равны, если равны поля title и author (Название и
Автор)
            return title == other.title && author == other.author;
        }

        public override string ToString()
        {
            return $"{author} - {title}";
        }
    }

    class Playlist : IEquatable<Playlist>
    {
        public string Name { get; private set; }
        public List<Song> Songs { get; private set; }

        public Playlist(string name)
        {
            Name = name;
            Songs = new List<Song>();
        }

        public void AddSong(Song song)
        {
            Songs.Add(song);
        }

        // Проверка, не содержит ли плейлист определенную песню
        public bool Contains(Song song)
        {
            foreach (Song tmpSong in Songs)
```

```
        {
            if (tmpSong.Equals(song)) return true;
        }

        return false;
    }

    public bool Equals(Playlist other)
    {
        // Плейлисты равны, когда равны все их песни между собой
        if (Songs.Count != other.Songs.Count) return false;

        for (int i = 0; i < Songs.Count; i++)
        {
            if (!Songs[i].Equals(other.Songs[i])) return false;
        }

        return true;
    }
}

public partial class Form1 : Form
{
    // Индекс «Плейлиста 1» в playlistComboBox
    const int PLAYLIST_1 = 0;

    private Playlist playlist1 = new Playlist("Плейлист 1");
    private Playlist playlist2 = new Playlist("Плейлист 2");

    public Form1()
    {
        InitializeComponent();
        playlistComboBox.SelectedIndex = PLAYLIST_1;
    }

    private void addSongButton_Click(object sender, EventArgs e)
    {
        string title = titleTextBox.Text;
        string author = authorTextBox.Text;
```

```
            Playlist playlist = playlistComboBox.SelectedIndex == PLAYLIST_1 ? playlist1 : playlist2;

            Song song = new Song(title, author);
            if (playlist.Contains(song))
            {
                MessageBox.Show($"Вы уже добавляли эту песню в {playlist.Name}.");
                return;
            }

            playlist.AddSong(song);

            if (playlistComboBox.SelectedIndex == PLAYLIST_1)
                playlist1ListBox.Items.Add(song);
            else
                playlist2ListBox.Items.Add(song);

            if (playlist1.Equals(playlist2))
                MessageBox.Show($"Плейлисты одинаковы. Вы могли бы удалить один из них.");
        }
    }
```

С результатами работы программы можно ознакомиться на рисунках ниже.

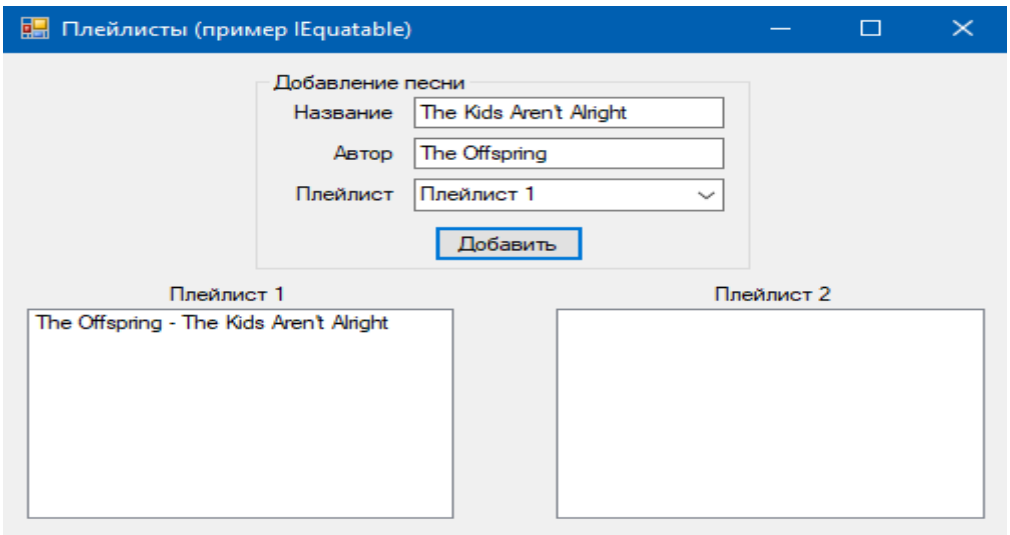

Рис. 29. Добавлена песня в «Плейлист 1»

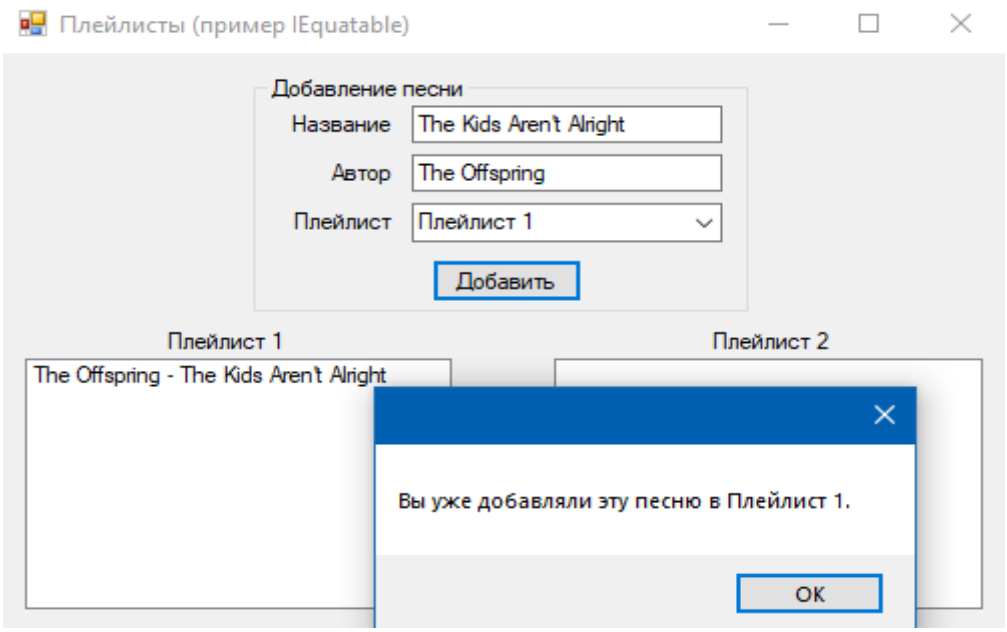

Рис. 30. Ошибка при попытке повторного добавления песни в плейлист

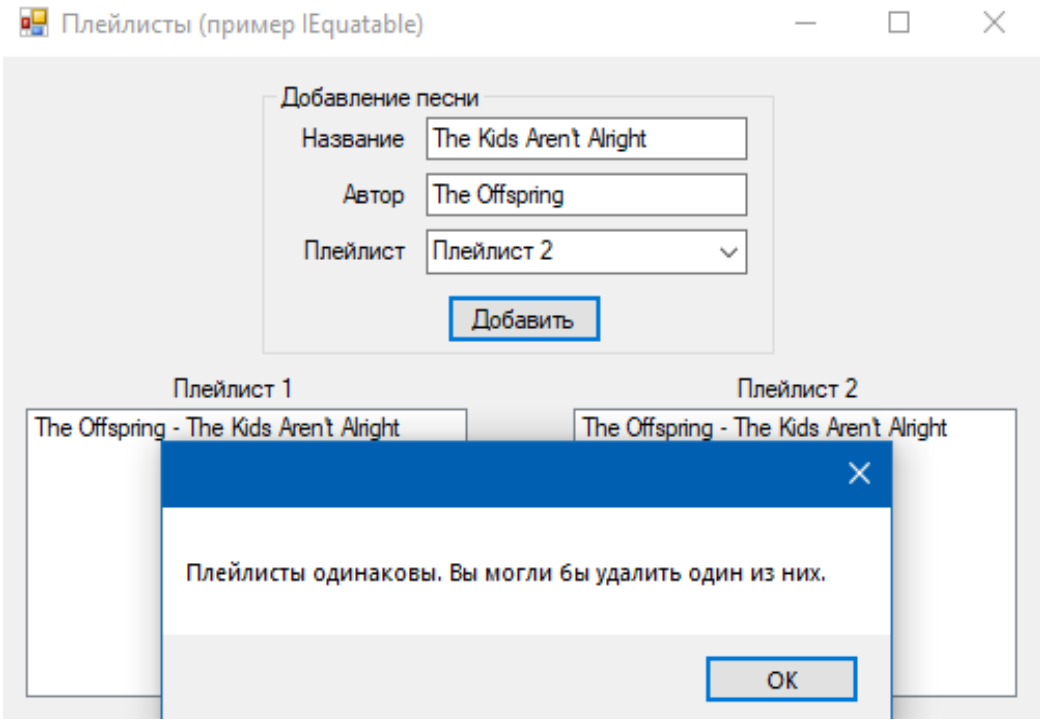

Рис. 31. Добавление той же песни в «Плейлист 2», уведомление о равенстве плейлистов

Проверка на равенство по разным критериям (Интерфейс IEqualityComparer)

IEqualityComparer находится в пространстве имен System.Collections. Принцип работы данного интерфейса схож с принципом IEquatable, однако IEqualityComparer позволяет производить проверку на равенство разными способами, для каждого создавая отдельный класс, реализующий этот интерфейс. Имеется всего один метод Equals, принимающий в качестве аргументов два сравниваемых объекта и возвращающий bool значение:

```
interface IEqualityComparer <T>
{
   bool Equals (T x, T y);
}
```

Перебор объектов (интерфейс IEnumerable)

Оператор *foreach* является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять перебор благодаря тому, что в них реализован интерфейс *IEnumerable и Enumerator*. Для применения оператора *foreach* к пользовательскому типу данных требуется реализовать в нем эти интерфейсы.

Интерфейс *IEnumerable* (перечисляемый) определяет всего один метод – *GetEnumerator*, возвращающий объект типа *IEnumerator* (перечислитель), который можно использовать для просмотра элементов объекта.

Интерфейс *IEnumerator* задает три элемента:

– свойство *Current*, возвращающий текущий элемент объекта;

– метод *MoveNext*, продвигающий перечислитель на следующий элемент объекта;

– метод *Reset*, устанавливающий перечислитель в начало просмотра.

Цикл foreach использует эти методы для перебора элементов класса, из которых состоит объект.

Пример использования интерфейса *IEnumerable* приведен в примере:

```
namespace IENUMERABLE
{
```

```
class ProgProgr : IEnumerator<int>
{
    int count;
    int position;
    int current;

    public ProgProgr(int count)
    {
        this.count = count;
        position = 0;
        current = 5;
    }
    public int Current
    {   get { return current; }   }
    object IEnumerator.Current
    {   get { return Current; }   }
    public void Dispose()
    {  }
    public bool MoveNext()
    {
        if (position > 0)
            current += 5;
        if (position < count)
        {
            position++;
            return true;
        }
        return false;
    }
    public void Reset()
    {
        position = 0;
        current = 5;
    }
}
class Progres : IEnumerable<int>
{
    int count;
    public Progres(int count)
    {           this.count = count;         }
    public IEnumerator<int> GetEnumerator()
```

```
        {       return new ProgProgr(count);    }
        IEnumerator IEnumerable.GetEnumerator()
        {       return GetEnumerator();         }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Progres a = new Progres(10);
            foreach (var x in a)
                Console.WriteLine(x);
            Console.ReadKey();
        }
    }
}
```

Результат работы программы:

```
5
10
15
20
25
30
35
40
45
50
```

Контрольные вопросы

1. Что называется интерфейсом?
2. Синтаксис объявления интерфейса.
3. Какие спецификаторы могут быть указаны для интерфейса?
4. Какой спецификатор имеет интерфейс по умолчанию?
5. Поддерживает ли интерфейс множественное наследование?
6. Содержит ли интерфейс поля, константы, конструкторы, деструкторы, статические элементы?
7. Какой спецификатор доступа имеют элементы интерфейса по умолчанию?

8. Задаются ли спецификаторы элементов интерфейса явным образом?

9. Может ли интерфейс кроме абстрактных содержать и обычные методы?

10. Все ли абстрактные методы интерфейса должны переопределяться в классе-потомке?

11. С какими спецификаторами доступа должны быть переопределены элементы интерфейса в производном классе?

12. Какими способами можно обратиться к элементам интерфейса, переопределённым в производном классе?

13. Если при реализации интерфейса в классе было явно указано имя интерфейса перед реализуемым элементом, нужно ли указывать спецификатор доступа, если нужно, то какой?

14. В каком случае к переопределённым элементам интерфейса можно обратиться только через объект типа интерфейса?

15. Класс наследник двух интерфейсов содержит метод с одним именем и одной и той же сигнатурой. Как обратиться к этим методам различных интерфейсов?

4 Делегаты, события

4.1 Перегрузка операторов

C# позволяет переопределить большинство операций так, чтобы при использовании их объектами конкретного класса выполнялись действия, отличные от стандартных. Это дает возможность применять объекты собственных типов данных в составе выражений, например:

```
newObject x, y, z;
…
z = x + y; // используется операция сложения, переопределенная для класса newObject
```

Определение собственных операций класса называют перегрузкой операций. Перегрузка операций обычно применяется для классов, для которых семантика операций делает программу более понятной. Если назначение операции интуитивно непонятно, перегружать такую операцию не рекомендуется.

Операции класса описываются с помощью методов специального вида, синтаксис которых выглядит следующим образом:

```
[ атрибуты] спецификаторы объявитель_операции
{
   тело
}
```

В качестве спецификаторов одновременно используются ключевые слова public и static. Кроме того, операцию можно объявить как внешнюю –extern. Объявление операции может выглядеть по-разному в зависимости от того, что мы перегружаем: унарную или бинарную операцию. Но в любом случае:

− операция должна быть описана как открытый статический метод класса (public static);

− параметры в операцию должны передаваться по значению (то есть недопустимо использовать параметры ref, out или params);

− допустимо определять несколько вариантов перегруженных операций класса, но их сигнатуры (заголовки) должны различаться; типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

4.2 Унарные операции

В классе можно переопределять следующие унарные операции:
+ - ! ~ ++ --, а также константы true и false.

Шилдт обращает наше внимание на то, что если была перегружена константа true, то должна быть перегружена и константа false и наоборот.

Синтаксис объявителя унарной операции:

тип operator унарная_операция (параметр)

Примеры заголовков унарных операций:

public static int operator + (DemoArray m)
public static DemoArray operator --(DemoArray m)
public static bool operator true (DemoArray m)

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. При этом операция должна возвращать:

– для операций +, -, !, ~ величину любого типа;

– для операций ++, -- величину типа класса, для которого она определяется;

– для операций true и false величину типа bool.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата. В качестве примера рассмотрим класс DemoArray, реализующий одномерный массив, в котором содержатся следующие функциональные элементы:

– конструктор, позволяющий создать объект-массив на основе последовательности чисел переменной длины;

– конструктор, позволяющий инициализировать объект-массив на основе уже существующего объекта-массива;

– метод, выводящий объект-массив на экран;

– свойство, возвращающее размерность массива;

– индексатор, позволяющий просматривать и устанавливать значение по индексу в закрытом поле-массиве;

– перегрузка операции унарный минус (все элементы массива меняют свое значение на противоположное);

– перегрузка операции инкремента (все элементы массива увеличивают свое значение на 1);

– перегрузка констант true и false (при обращении к объекту будет возвращаться значение true, если все элементы массива положительные, в противном случае будет возвращаться значение false).

```
using System;

namespace MyProgram
{
   class DemoArray
  {
     int[] MyArray;

     public DemoArray(params int []array) //конструктор 1
    {
       MyArray = new int[array.Length];
       Array.Copy(array,MyArray, array.Length);
     }
     public DemoArray(DemoArray array) //конструктор 2
    {
       MyArray = new int[array.Length];
       Array.Copy(array.MyArray, MyArray, array.Length);
     }

     public void Show()
    {
       foreach (int item in MyArray)
      {
         Console.Write("{0} ", item);
      }
      Console.WriteLine();
    }

     public int Length //свойство, возвращающее размерность массива
    {
       get  { return MyArray.Length; }
     }
     public int this[int i] //индексатор
    {
       get
      {
         if (i = MyArray.Length)
```

```
            {
                Console.WriteLine("Индекс {0} выходит за границы массива", i);
                return 0;
            }
            else  return MyArray[i];
        }
        set
        {
            if (i = MyArray.Length)
            {
                Console.WriteLine("Индекс {0} выходит за границы массива", i);
            }
            else  MyArray[i] = value;
        }
    }

    public static DemoArray operator - (DemoArray x) //перегрузка операции
унарный минус
    {
        DemoArray temp = new DemoArray(x);
        for (int i = 0; i < x.Length; i++)
            temp[i] = -x[i];
        return temp;
    }

    public static DemoArray operator ++ (DemoArray x) //перегрузка операции
инкремента
    {
        DemoArray temp = new DemoArray(x);
        for (int i = 0; i < x.Length; i++)
            temp[i] = x[i]+1;
        return temp;
    }

    public static bool operator true(DemoArray a) //перегрузка константы true
    {
        foreach (int item in a.MyArray)
        {
            if (item<0) return true;
        }
        return false;
```

```
  }
}

  public static bool operator false(DemoArray a) //перегрузка константы false
 {
    foreach (int item in a.MyArray)
    {
       if (item>0) return true;
    }
    return false;
 }
}
}
```

Продемонстрируем работу с данным классом:

```
using System;

namespace MyProgram
{
   class Program
  {
     static void Main()
     {
        DemoArray oneArray = new DemoArray(1, -4, 3, -5, 0);
        Console.Write("Первый массив: ");
        oneArray.Show();
        Console.WriteLine();

        Console.WriteLine("Унарный минус");
        DemoArray twoArray=-oneArray;
        Console.Write("Первый массив: ");
        oneArray.Show();
        Console.Write("Второй массив ");
        twoArray.Show();
        Console.WriteLine();

        Console.WriteLine("Операция префиксного инкремента");
        ++oneArray; //1
        Console.Write("Первый массив: ");
        oneArray.Show();
```

```
        Console.WriteLine();

        Console.WriteLine("Операция постфиксного инкремента");
        DemoArray threeArray=oneArray++; //2
        Console.Write("Первый массив: ");
        oneArray.Show();
        Console.Write("Третий массив: ");
        threeArray.Show();
        Console.WriteLine();

        //проверка на положительность элементов массива
       if (oneArray) Console.WriteLine("В первом массиве все элементы
положительные");
        else Console.WriteLine("В первом массиве есть не положительные
элементы");
     }
   }
}
```

Результат работы программы:
Первый массив: 1 -4 3 -5 0
Унарный минус
Первый массив: 1 -4 3 -5 0
Второй массив: -1 4 -3 5 0
Операция префиксного инкремента
Первый массив: 2 -3 4 -4 1
Операция постфиксного инкремента
Первый массив: 3 -2 5 -3 2
Третий массив: 2 -3 4 -4 1
В первом массиве есть не положительные элементы

Обратите внимание на то, что перегруженную операцию инкремента можно использовать как в качестве самостоятельной операции (строка 1), так и в составе выражения (строка 2).

4.3 Бинарные операции

При разработке класса можно перегрузить следующие бинарные операции:

+ - * / % & | ^ << >> == != < > <= >= .

Обратите внимание – операций присваивания в этом списке нет.
Синтаксис объявителя бинарной операции:

тип operator бинарная_операция (параметр1, параметр 2)

Примеры заголовков бинарных операций:

public static DemoArray operator + (DemoArray a, DemoArray b)

public static bool operator == (DemoArray a, DemoArray b)

При переопределении бинарных операций нужно учитывать следующие правила:

– хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется.

– операция может возвращать величину любого типа.

– операции отношений определяются только парами и обычно возвращают логическое значение. Чаще всего переопределяются операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение значения некоторых полей объектов, а не ссылок на объект. Для того чтобы переопределить операции отношений, требуется знание стандартных интерфейсов.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него две версии переопределенной операции +:

Вариант 1: добавляет к каждому элементу массива заданное число;

Вариант 2: поэлементно складывает два массива.

```
public static DemoArray operator +(DemoArray x, int a) //вариант 1
{
   DemoArray temp = new DemoArray(x);
   for (int i = 0; i < x.Length; ++i)
  {
      temp[i]=x[i]+a;
  }
   return temp;
}
public static DemoArray operator +(DemoArray x, DemoArray y) //вариант 2
{
   if (x.Length == y.Length)
  {
     DemoArray temp = new DemoArray(x);
     for (int i = 0; i < x.Length; i++)
    {
```

```
            temp[i] = x[i] + y[i];
        }
        return temp;
    }
    else
    {
        Console.WriteLine("Несоответствие размерностей массивов");
        return null;
    }
}
```

Рассмотрим на фрагменте программы пример использования перегруженной операции сложения:

```
DemoArray oneArray = new DemoArray(1, -4, 3, -5, 0);
DemoArray twoArray = new DemoArray(3, 6, 10, 0, -2);
DemoArray threeArray = oneArray+twoArray;
DemoArray fourArray = oneArray+5;
Console.Write("Первый массив: ");
oneArray.Show();
Console.Write("Второй массив: ");
twoArray.Show();
Console.Write("Третий массив: ");
threeArray.Show();
Console.Write("Четвертый массив: ");
fourArray.Show();
```

Результат работы фрагмента программы:
Первый массив: 1 -4 3 -5 0
Второй массив: 3 6 10 0 -2
Третий массив: 4 2 13 -5 -2
Четвертый массив: 6 1 8 0 5

4.4 Операции преобразования типов

Операции преобразования типов обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных. Синтаксис описания операции преобразования типов выглядит следующим образом:

explicit operator целевой_тип (параметр) //явное преобразование
implicit operator целевой_тип (параметр) //неявное преобразование

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого выполняется преобразование.

Неявное преобразование выполняется автоматически в следующих ситуациях:

– при присваивании объекта переменной целевого типа;

– использовании объекта в выражении, содержащем переменные целевого типа;

– передаче объекта в метод параметра целевого типа;

– явном приведении типа.

Явное преобразование выполняется при использовании операции приведения типа.

При определении операции преобразования типа следует учитывать следующие особенности:

– тип возвращаемого значения (целевой_тип) включается в сигнатуру объявителя операции;

– ключевые слова explicit и implicit не включаются в сигнатуру объявителя операции.

Следовательно, для одного и того же класса нельзя определить одновременно и явную, и неявную версию. Однако, т.к. неявное преобразование автоматически выполнятся при явном использовании операции приведения типа, то, если необходимо, чтобы для проектируемого класса выполнялись обе операции преобразования, достаточно разработать только неявную версию операции.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него неявную версию переопределения типа DemoArray в тип одномерный массив и наоборот:

```
//неявное преобразование типа int [] в DemoArray
public static implicit operator DemoArray (int []a)
{
   return new DemoArray(a);
}
//неявное преобразование типа DemoArray в int []
public static implicit operator int [](DemoArray a)
{
   int [] temp = new int[a.Length];
   for (int i = 0; i < a.Length; i++)
```

```
  {
     temp[i] = a[i];
  }
   return temp;
}
```

Продемонстрируем работу данных методов на примере:

```
using System;

namespace MyProgram
{
   class Program
   {
      static void Print(int[]a) //выводит на экран одномерный массив
      {
         for (int i = 0; i < a.Length; i++)
         Console.Write(a[i] + " ");
         Console.WriteLine();
      }

      static void Main()
      {
         DemoArray a = new DemoArray(1, -4, 3, -5, 0);
         int [] mas1 = a; //неявное преобразование типа DemoArray в int []
         int [] mas2 = (int []) a; //явное преобразование типа DemoArray в int []
         DemoArray b1 = mas1; //неявное преобразование типа int [] в DemoArray
         DemoArray b2 = (DemoArray) mas2;//явное преобразование типа int [] в DemoArray

         //изменение значений
         mas1[0] = 0; mas2[0] = -1;
         b1[0] = 100; b2[0] = -100;

         //вывод данных на экран
         Console.Write("Массива a: ");
         a.Show();
         Console.Write("Массив mas1: ");
```

```
            Print(mas1);
            Console.Write("Массив mas2: ");
            Print(mas2);
            Console.Write("Массив b1: ");
            b1.Show();
            Console.Write("Массив b2: "); b2.Show();
        }
    }
}
```

Результат работы программы:
Массив a: 1 -4 3 -5 0
Массив mas1: 0 -4 3 -5 0
Массив mas2: -1 -4 3 -5 0
Массив b1: 100 -4 3 -5 0
Массив b2: -100 -4 3 -5 0

4.5 Делегаты

Делегат – это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка. Рассмотрим сначала второй случай. Приведенное Павловской описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

[атрибуты] [спецификаторы] delegate тип имя_делегата ([параметры])

Спецификаторы делегата имеют тот же смысл, что и для класса, причем допускаются только спецификаторы new, public, protected, internal и private.

Тип описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательным параметрами делегата являются параметры этих методов.

Пример описания делегата:

```
public delegate void D ( int i );
```

Здесь описан тип делегата, который может хранить ссылки на методы, возвращающие void и принимающие один параметр целого типа.

Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

Использование делегатов

Для того чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы.

Делегаты применяются в основном для следующих целей:

– получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;

– обеспечения связи между объектами по типу ≪источник – наблюдатель≫;

– создания универсальных методов, в которые можно передавать другие методы;

– поддержки механизма обратных вызовов.

Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

Добавление метода в список выполняется либо с помощью метода Combine, унаследованного от класса System.Delegate, либо, что удобнее, с помощью перегруженной операции сложения.

Простейшее использование делегата:

```
using System;
using System.Windows.Forms;

namespace delegates_1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();

         Greeting greeting; // Создание переменной
         greeting = ShowGoodDay; // Присваивание ссылки на метод
         greeting += ShowTime; // Добавление ссылки на другой метод
```

```
            greeting("Гость"); // Вызов всех методов делегата
        }

        private delegate void Greeting(string name);

        private static void ShowGoodDay(string name)
        {
            MessageBox.Show($"Добрый день, {name}!");
        }

        private static void ShowTime(string name)
        {
            MessageBox.Show($"{name},
{DateTime.Now.ToShortTimeString()}");
        }
    }
    …
}
```

При запуске программы будет отображаться следующее:

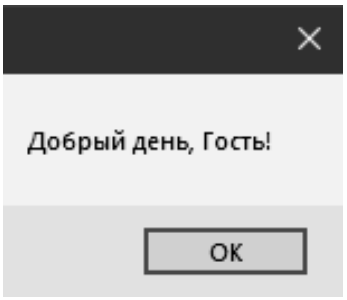

Рис. 32. Приветствие гостя

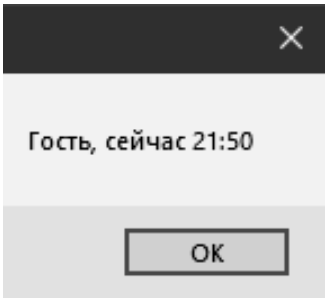

Рис. 33. Оглашение времени после приветствия

Ниже представлен второй пример. Здесь делегат используется для указания, в каком порядке сортировать массив – по возрастанию или убыванию.

```
using System;
using System.Windows.Forms;

namespace delegates_2
{
   public partial class Form1 : Form
   {
      // Номер сортировки по возрастанию (индекс в списке)
      const int ASCENDING_SORT = 0;

      public Form1()
      {
         InitializeComponent();
         sortBy.SelectedIndex = ASCENDING_SORT;
      }

      // Делегат для сравнивания двух чисел
      private delegate bool CompareNumbers(double num1, double num2);

      private static bool IsGreater(double num1, double num2)
      {
         return num1 > num2;
      }

      private static bool IsLess(double num1, double num2)
      {
         return num1 < num2;
      }

      // Сортировка массива методом вставки
      // Если compare ссылается на isGreater, то сортировка по возрастанию
      // Если compare ссылается на isLess, то сортировка по убыванию
      private void SortNumberArray(double[] array, CompareNumbers compare)
      {
```

```
            for (int i = 1; i < array.Length; i++)
            {
                double key = array[i];
                int j = i - 1;

                while (j >= 0 && compare(array[j], key))
                {
                    array[j + 1] = array[j];
                    j -= 1;
                }

                array[j + 1] = key;
            }
        }

        // По нажатию кнопки для сортировки введенного массива
        private void sortButton_Click(object sender, EventArgs e)
        {
            // Инициализируем введенный массив
            string[] tmp = arrayTextBox.Text.Split('\n');
            double[] array = new double[tmp.Length];
            for (int i = 0; i < tmp.Length; i++)
                double.TryParse(tmp[i], out array[i]);

            // Создаем делегат, который будет определять порядок сортировки
            CompareNumbers compare;
            if (sortBy.SelectedIndex == ASCENDING_SORT)
                compare = new CompareNumbers(IsGreater);
            else
                compare = new CompareNumbers(IsLess);

            // Вызываем метод сортировки, передавая в его аргументах делегат
            SortNumberArray(array, compare);

            // Выводим сортированный массив
            sortedArrayTextBox.Text = "";
            foreach (double number in array)
                sortedArrayTextBox.Text += $"{number}\r\n";
        }
```

```
    }
    …
}
```

Результаты работы программы можно увидеть на рисунках ниже.

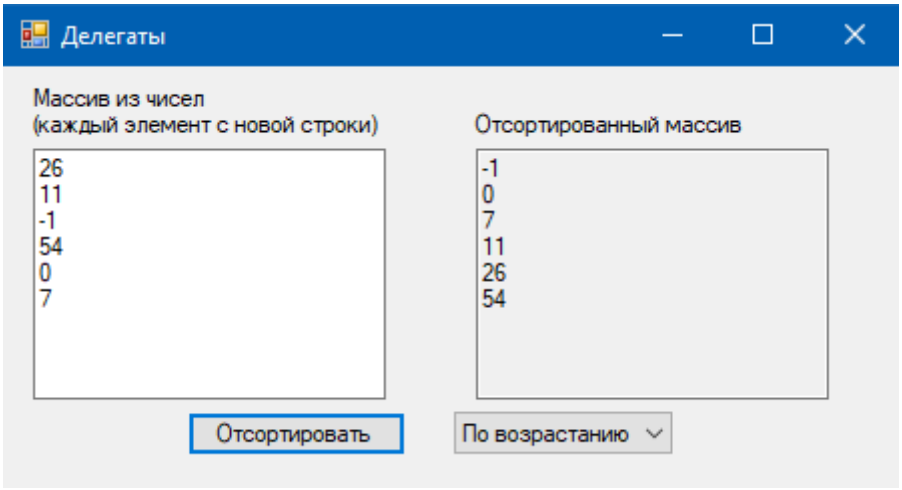

Рис. 34. Сортировка массива по возрастанию

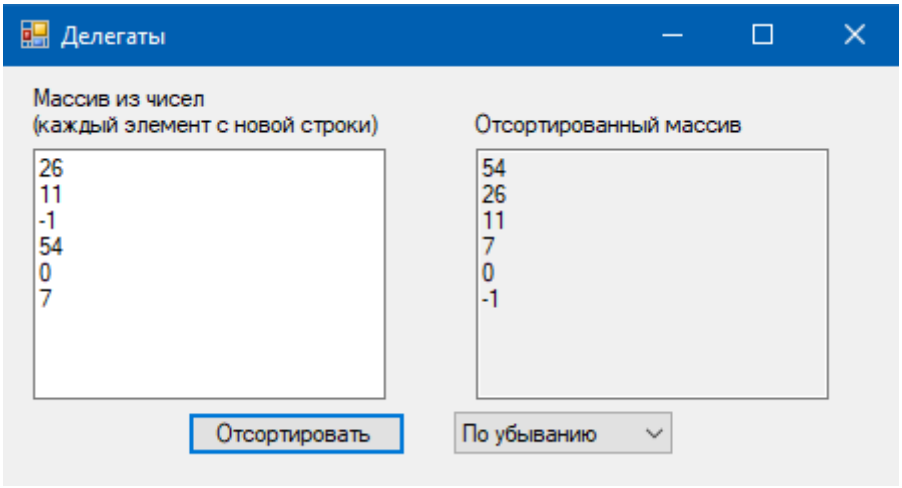

Рис. 35. Сортировка массива по убыванию

И третий пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace 10.1
{
   delegate void call(ref string s); //объявление делегата
   class Program
   {
      static void Main(string[] args)
      {
         string text = "Hello world";
         call c; // экземпляр делегата
         for (int i = 0; i < 2; i++)
         {
            c = new call(Hello); // инициализация методом Hello
            if (i==1)
            {
               c = new call(World); // инициализация методом World
            }
            c(ref text);
            Console.WriteLine(text);

         }
         Console.ReadKey();
      }
      public static void Hello(ref string text)
      {
         string buffer = "";
         for (int i = 0; i < text.Length; i++)
         {
            if ((text[i]=='l') || text[i]=='L')
            {
               buffer += '1';
            }
            else if (text[i]=='H')
            {
               buffer += "|-|";
            }
            else
            {
               buffer += text[i];
            }
```

```
            }
            text = buffer;
        }
        public static void World(ref string text)
        {
            string buffer = "";
            for (int i = 0; i < text.Length; i++)
            {
                if (i/2*2==i)
                {
                    buffer += char.ToUpper(text[i]);
                }
                else
                {
                    buffer += text[i];
                }
            }
            text = buffer;
        }
    }
}
```

Вот как выглядит измененный метод Main, в котором одним вызовом делегата выполняется преобразование исходной строки сразу двумя методами:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _10._2
{
    public delegate void call(object o);// объявление делегата
    class Program
    {
        static void Main(string[] args)
        {
            Source s = new Source(); // объект класса-источника
            BeholderOne b1 = new BeholderOne(); // объекты
```

```
            BeholderTwo b2 = new BeholderTwo(); // класс-наблюдателя
            s.Del(new call(b1.Speech)); // регистрация методов
            s.Del(new call(b2.Speech)); // наблюдателей в источнике
            s.Trigger(); // инициализирование события
            Console.ReadKey();
        }
    }
    class Source // класс-источник
    {
        call c;
        public void Del(call C) // регистрация делегата
        {
            c += C;
        }
        public void Trigger() // метод-событие
        {
            Console.WriteLine("enter complited"); // оповещение наблюдателей
            if (c != null)
            {
                c(this);
            }
        }
    }
    class BeholderOne //класс-наблюдатель
    {
      public void Speech(object o)
      {
          Console.WriteLine("reaction niticed ONE"); //реакция на событие
      }
    }
    class BeholderTwo //класс-наблюдатель
    {
        public void Speech(object o)
        {
            Console.WriteLine("reaction niticed TWO"); //реакция на событие
        }
    }
}
```

При вызове последовательности методов с помощью делегата необходимо учитывать следующее:

– сигнатура методов должна в точности соответствовать делегату;

– методы могут быть как статическими, так и обычными методами класса;

– каждому методу в списке передается один и тот же набор параметров;

– если параметр передается по ссылке, изменения параметра в одном методе отразятся на его значении при вызове следующего метода;

– если параметр передается с ключевым словом си или метод возвращает значение, результатом выполнения делегата является значение, сформированное последним из методов списка (в связи с этим рекомендуется формировать списки только из делегатов, имеющих возвращаемое значение типа void);

– если в процессе работы метода возникло исключение, не обработанное в том же методе, последующие методы в списке не выполняются, а происходит поиск обработчиков, объемлющих делегат блоках;

– попытка вызвать делегат, в списке которого нет ни одного метода, вызывает генерацию исключения. System.NullReferenceExeption.

Паттерн «наблюдатель»

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов. При этом желательно избежать жесткой связанности классов, так как это часто негативно сказывается на возможности многократного использования кода, напоминает нам Шилдт.

Наблюдатель (observer) определяет между объектами зависимость типа ≪один ко многим≫, так что при изменении состоянии одного объекта все зависящие от него объекты получают извещение и автоматически обновляются.

Оповещение наблюдателей с помощью делегата:

```
using System;
namespace ConsoleApplication
{
   public delegate void Del( object o );   // объявление делегата
   class Subj                              // класс-источник
   {
      Del dels;                            // объявление экземпляра делегата
      public void Register( Del d )        // регистрация делегата
```

```
        {
            dels += d;
         }
         public void Hello ()  // что-то произошло
        {
           Console.WriteLine ("Привет!");
           if ( dels != null ) dels( this );    // оповещение наблюдателей
        }
     }
     class ObsA     // класс-наблюдатель
    {
       public void Do( object o )            // реакция на событие источника
      {
          Console.WriteLineC "Вижу, что Привет!");
      }
    }
    class ObsB                                          // класс-наблюдатель
   {
      public static void See ( object o )            // реакция на событие источника
     {
         Console.WriteLine ( "Я тоже вижу, что Привет!" );
      }
   }
   class Classl
  {
     static void Main()
    {
       Subj s = new Subj();             // объект класса-источника
       ObsA ol = new ObsA();       // объекты
       ObsA o2 - new ObsAO;        // класса-наблюдателя
       s.Register( new Del( ol.Do ) );  // регистрация методов
       s.Register( new Del( o2.Do ) );  // наблюдателей в источнике
       s.Register( new Del( ObsB.See ) );  // ( экземпляры делегата )
       s.Hello();                          // инициирование события
}
}
```

Результат работы программы:
Привет!
Вижу, что Привет!
Я тоже вижу, что Привет!

Операции

Делегаты можно сравнивать на равенство и неравенство. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке. Делегаты, различающиеся только именами, считаются имеющими разные типы. С делегатами одного типа можно выполнять операции простого и сложного присваивания.

Делегат, как и строка string, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

С делегатами одного типа можно выполнять операции простого и сложного присваивания, например:

```
Del dl = new Del( ol.Do ):          // ol.Do
Del d2 = new Del( o2.Do );          // o2.Do
Del d3 = dl + d2;                   // ol.Do и o2.Do
d3 += d l :                         // ol.Do. o2.Do и ol.Do
d3 -= d2;                           // Ol.Do и ol.Do
```

Передача делегатов в методы

Поскольку делегат является классом, его можно передавать в методы в качестве параметра. Таким образом обеспечивается функциональная параметризация: в метод можно передавать не только различные данные, но и различные функции их обработки. Функциональная параметризация применяется для создания универсальных методов и обеспечения возможности обратного вызова.

Обратный вызов (callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра.

Механизм обратного вызова широко используется в программировании. Например, он реализуется во многих стандартных функциях Windows.

Передача делегата через список параметров

```
namespace _10._3
{
   class Program
   {
      public delegate double call(double x);
      static void Main(string[] args)
```

```
        {
            Console.WriteLine("Sin values:");
            Table(new call(Math.Sin), -2, 2);
            Console.WriteLine("Cos values:");
            Table(new call(Math.Cos), -2, 2);
            Console.ReadKey();
        }
        public static void Table(call c, double x, double b)
        {
            for (double i = x; i <= b; i++)
            {
                Console.WriteLine($"X={x}, Y={c(x)}");
            }
        }
    }
}
```

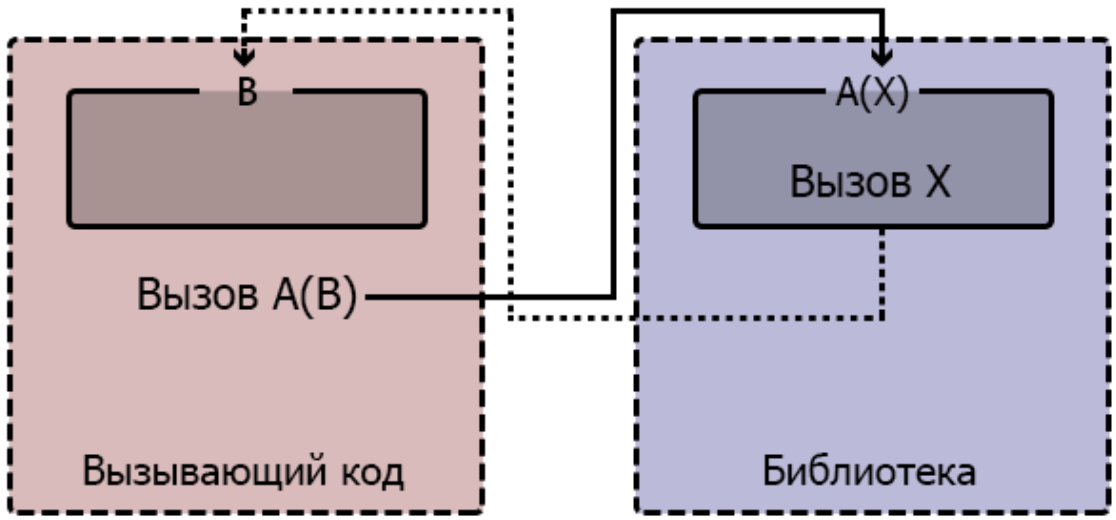

Рис. 36. Механизм обратного вызова

Результат работы программы:

Таблица функции Sin
X=-2, Y=-0,9092974268256817
X=-2, Y=-0,9092974268256817
X=-2, Y=-0,9092974268256817
X=-2, Y=-0,9092974268256817
X=-2, Y=-0,9092974268256817

Таблица функции Cos
X=-2, Y=-0,4161468365471424
X=-2, Y=-0,4161468365471424
X=-2, Y=-0,4161468365471424
X=-2, Y=-0,4161468365471424
X=-2, Y=-0,4161468365471424

Обработка исключений при вызове делегатов

Ранее говорилось о том, что если в одном из методов списка делегата генерируется исключение, следующие методы не вызываются. Этого можно избежать, если обеспечить явный перебор всех методов в проверяемом блоке и обрабатывать возникающие исключения. Все методы, заданные в экземпляре делегата, можно получить с помощью унаследованного метода GetInvocationList.

Свойство Method возвращает результат типа Method Info. Класс Method Info содержит множество свойств и методов, позволяющих получить полную информацию о методе, например его спецификаторы доступа, имя и тип возвращаемого значения.

Перехват исключений при вызове делегата:

```
using System;

namespace _10._6
{
   delegate void call(ref string s); //объявление делегата
   class Program
   {
      static void Main(string[] args)
      {
         string text = "Hello world";
         call c = new call(Hello); // экземпляр делегата
         c += new call(BadWorld); // дополнение списка методов
         c += new call(World); // дополнение списка методов
         foreach (call fun in c.GetInvocationList())
         {
            try
            {
               fun(ref text); // вызов метода из списка
            }
            catch (Exception e)
            {
```

```
            Console.WriteLine(e.Message);
            Console.WriteLine("Exeption in method " + fun.Method.Name);
        }
    }
    Console.WriteLine("Result - "+text);
    Console.ReadKey();
}
public static void Hello(ref string text)
{
    string buffer = "";
    for (int i = 0; i < text.Length; i++)
    {
        if ((text[i] == 'l') || text[i] == 'L')
        {
            buffer += '1';
        }
        else if (text[i] == 'H')
        {
            buffer += "|-|";
        }
        else
        {
            buffer += text[i];
        }
    }
    text = buffer;
}
public static void World(ref string text)
{
    string buffer = "";
    for (int i = 0; i < text.Length; i++)
    {
        if (i / 2 * 2 == i)
        {
            buffer += char.ToUpper(text[i]);
        }
        else
        {
            buffer += text[i];
        }
    }
```

```
            text = buffer;
        }
        public static void BadWorld(ref string text)
        {
            Console.WriteLine(" Method called BadWorld");
            throw new Exception();
        }
    }
}
```

Результат работы программы:

```
вызван метод Hello
вызван метод BadWorld
Exception of type System.Exception was thrown.
Exception in method BadWorld
вызван метод World
результат - Hello world
```

4.6 События

Событие является элементом класса, который позволяет ему посылать другим объектам уведомления об изменении своего состояния. Заметим, что для объектов, являющихся наблюдателями события, активизируются методы-обработчики этого события. Обработчики должны быть зарегистрированы в объекте-источнике события. Таким образом, механизм событий формализует на языковом уровне паттерн «наблюдатель», который рассматривался в предыдущем разделе.

Механизм событий можно также описать с помощью модели «публикация-подписка»: один класс, являющийся отправителем (sender) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся получателями (receivers) сообщения, подписываются на получение этих событий.

События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

- описания делегата, задающего сигнатуру обработчиков событий;
- описания события;
- описания метода (методов), инициирующих событие.

Синтаксис события похож на синтаксис делегата:

[атрибуты] [спецификаторы] event тип имя_события

Для событий применяются спецификаторы new, pub1ic, protected, internal, private, static, virtual, sealed, override, abstract и extern, которые изучались при рассмотрении методов классов. Павловская приводит пример: так же, как и методы, событие может быть статическим (static), тогда оно связано с классом в целом, или обычным – в этом случае оно связано с экземпляром класса.

Тип события – это тип делегата, на котором основано событие.

Пример описания делегата, на котором основано событие.

```
public deledate void Del (object o);  // объявление делегата
class Example
{
    public event Del Oops;              // объявление события
    …
}
```

Обработка событий выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс), который желает получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.

Это в точности тот же самый механизм, который рассматривался в предыдущем разделе. Единственное отличие состоит в том, что при использовании событий не требуется описывать метод, регистрирующий обработчики, поскольку события поддерживают операции += и -=, добавляющие обработчик в список и удаляющие его из списка.

Событие состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.

Пример оповещения наблюдателей с помощью событий приведен ниже.

```
using System;
namespace _10._7
{
   public delegate void call();// объвление делегата
   class Program
   {
      static void Main(string[] args)
      {
```

```
        Source f = new Source(); // объект класса-источника

        BeholderOne b1 = new BeholderOne(); // объекты
        BeholderOne b2 = new BeholderOne(); // класс-наблюдателя

        f.c += new call(b1.Speech);          // добавление
        f.c += new call(b2.Speech);          // обработчиков
        f.c += new call(BeholderTwo.Speech2); // к событию

        f.Trigger(); // инициализирование события
        Console.ReadKey();       }
}
class Source // класс-источник
{

    public event call c;

    public void Trigger()
    {
        Console.WriteLine("enter complited");
        if (c != null)
        {
            c();
        }
    }
}
class BeholderOne //класс-наблюдатель
{
    public void Speech()
    {
```

```
        Console.WriteLine("reaction niticed ONE"); //реакция на событие
      }
   }
   class BeholderTwo //класс-наблюдатель
   {
      public static void Speech2()
      {
         Console.WriteLine("reaction niticed TWO"); //реакция на событие
      }
   }
}
```

Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции += и -=. Тип результата этих операций – void, в отличие от операций сложного присваивания для арифметических типов. Иного способа доступа к списку обработчиков нет.

Внутри класса, в котором описано событие, с ним можно обращаться, как с обычным полем, имеющим тип делегата: использовать операции отношения, присваивания и т. д. Значение события по умолчанию – null. Например, в методе CryOops выполняется проверка на null для того, чтобы избежать генерации исключения System.NullbeferenceExcception.

В библиотеке .NET описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

– имя делегата заканчивается суффиксом EventHandler;

– делегат получает два параметра: первый параметр задает источник события и имеет тип object, второй параметр задает аргументы события и имеет тип EventArgs или производный от него.

Если обработчикам события требуется специфическая информация о событии, то для этого создают класс, производный от стандартного класса EventArgs, и добавляют в него необходимую информацию. Если делегат не использует такую информацию, можно не описывать делегата и собственный тип аргументов, а обойтись стандартным классом делегата System. EventHandler. Имя обработчика события принято составлять из префикса On и имени события. В листинге 10.8 приведен пример из

листинга 10.7, оформленный в соответствии со стандартными соглашениями NET.

```
using System;

namespace 10.8
{
   class Program
   {
      static void Main(string[] args)
      {
         Source f = new Source(); // объект класса-источника

         BeholderOne b1 = new BeholderOne(); // объекты
         BeholderOne b2 = new BeholderOne(); // класс-наблюдателя

         f.c += new EventHandler(b1.Speech);              // добавление
         f.c += new EventHandler(b2.Speech);              // обработчиков
         f.c += new EventHandler(BeholderTwo.Speech); // к событию
         f.Trigger(); // инициализирование события
         Console.ReadKey();
      }
   }
   class Source // класс-источник
   {

      public event EventHandler c;

      public void Trigger()
      {
         Console.WriteLine("enter complited");
         if (c != null)
         {
            c(this, null);
         }
      }
   }
   class BeholderOne //класс-наблюдатель
   {
      public void Speech(object sender, EventArgs e)
      {
```

```
        Console.WriteLine("reaction niticed ONE"); //реакция на событие
    }
  }
  class BeholderTwo //класс-наблюдатель
  {
    public static void Speech(object sender, EventArgs e)
    {
      Console.WriteLine("reaction niticed TWO"); //реакция на событие
    }
  }
}
```

Те, кто работает с C# версии 2.0, могут упростить эту программу, используя новую возможность неявного создания делегатов при регистрации обработчиков событий. Соответствующий вариант приведен в листинге 10.9. В демонстрационных листингах добавлен новый анонимный обработчик, появившийся в новой версии языка.

10.9 листинг

```
using System;


namespace ConsoleAppFor
{
  class Subj
  {
    public event EventHandler Hello;


    public void SayHello()
    {
      Console.WriteLine("Всем привет!");
      if (Hello != null) Hello(this, null);
    }
  }
  class Person1
  {
```

```
    public void OnHello(object sender, EventArgs e)
    {
        Console.WriteLine("Привет!");
    }
}
class Person2
{
    public static void OnHello(object sender, EventArgs e)
    {
        Console.WriteLine("Приветствую!");
    }
}
class Program
{
    static void Main()
    {
        Subj s = new Subj();
        Person1 o1 = new Person1();
        s.Hello += o1.OnHello;
        s.Hello += Person2.OnHello;
        s.Hello += delegate(object sender, EventArgs e)
        {
            Console.WriteLine("О, здравствуйте!");
        }
        s.SayHello();
    }
}
}
```

Контрольные вопросы

1. Дать определение делегату.
2. Синтаксис объявления делегата.
3. Перечислить спецификаторы делегата.
4. Где нужно размещать объявление делегата?
5. Для каких целей применяются делегаты?
6. В каком порядке вызываются ссылки, добавленные в делегат?
7. Перечислить правила вызова последовательности методов с помощью делегата.
8. Пояснить стратегию паттерна «наблюдатель».
9. Каким образом обеспечивается функциональная параметризация?
10. Пояснить механизм обратного вызова.

5 Сборки, библиотеки, атрибуты

5.1 Сборки

Сборка представляет собой файл с расширением exe или dll, который содержит код на промежуточном языке, метаданные типов, манифест и ресурсы.

Промежуточный язык (Intermediate Language, IL) не содержит инструкций, зависящих от операционной системы и типа компьютера, что обеспечивает две основные возможности:

– выполнение приложения на любом типе компьютера, для которого существует среда выполнения CLR;

– повторное использование кода, написанного на любом NET-совместимом языке.

IL-код можно просмотреть с помощью дизассемблера ILDasm.exe, который находится в папке ..SDK\bin\ каталога размещения Visual Studio. NET. После запуска ILDasm можно открыть любой файл среды .NET с расширением exe или dll с помощью команды File -> Open. В окне программы откроется список всех элементов сборки, сведения о каждом можно получить двойным щелчком. При этом открывается окно, в котором для методов выводится доступный для восприятия дизассемблированный код.

Метаданные типов – это сведения о типах, используемых в сборке. Компилятор создаёт метаданные автоматически. В них содержится информация о каждом типе, имеющемся в программе, и о каждом его элементе. Например, для каждого класса описываются все его поля, методы, свойства, события, базовые классы и интерфейсы.

Среда выполнения использует метаданные для поиска определений типов и их элементов в сборке, для создания экземпляров объектов, проверки вызова методов и т. д. Компилятор, редактор кода и средства отладки также широко используют метаданные, например для вывода подсказок и диагностических сообщений.

Манифест – это набор метаданных о самой сборке, включая информацию обо всех файлах, входящих в состав сборки, версии сборки, а также сведения обо всех внешних сборках, на которые она ссылается. Манифест создается компилятором автоматически, программист может дополнять его собственными атрибутами.

Чаше всего сборка состоит из единственного файла, однако она может включать и несколько физических файлов (модулей). В этом случае манифест либо включается в состав одного из файлов, либо содержится в отдельном файле.

Многофайловые сборки используются для ускорения загрузки приложения – это имеет смысл для сборок большого объема, работа с которыми производится удаленно.

На логическом уровне сборка представляет собой совокупность взаимосвязанных типов – классов, интерфейсов, структур, перечислений, делегатов и ресурсов. Библиотека NET представляет собой совокупность сборок, которую используют приложения. Точно так же можно создавать и собственные сборки, которые можно будет задействовать либо в рамках одного приложения (частные сборки), либо совместно различными приложениями (открытые сборки). По умолчанию все сборки являются частными.

Манифест сборки содержит:

– идентификатор версии;

– список всех внутренних модулей сборки;

– список внешних сборок, необходимых для нормального выполнения сборки;

– информацию о естественном языке, используемом в сборке (например, русском);

– «сильное» имя (strong name) – специальный вариант имени сборки, используемый для открытых сборок;

– необязательную информацию, связанную с безопасностью;

– необязательную информацию, связанную с хранением ресурсов внутри сборки.

Идентификатор версии относится ко всем элементам сборки. Он позволяет избегать конфликтов имен и поддерживать одновременное существование и использование различных версий одних и тех же сборок. Идентификатор версии состоит из двух частей: информационной версии в виде текстовой строки и версии совместимости в виде четырех чисел, разделенных точками:

– основной номер версии (major version);

– дополнительный номер версии (minor version);

– номер сборки (build number);

– номер ревизии (revision number).

Среда выполнения применяет идентификатор версий для определения того, какие из открытых сборок совместимы с требованиями клиента. Например, если клиент запрашивает сборку 3.1.0.0, а присутствует только версия 3.4.0.0, сборка не будет опознана как подходящая, поскольку считается, что в дополнительных версиях могут произойти изменения в типах и их элементах. Разные номера ревизии допускают, но не гарантируют совместимость. Номер сборки на

совместимость не влияет, так как чаще всего он изменяется при установке заплатки, или патча (patch).

Идентификатор версии формируется автоматически, но при желании можно задать его вручную с помощью атрибута [AssemblyVersion], который рассматривается далее.

Информация о безопасности позволяет определить, предоставить ли клиенту доступ к запрашиваемым элементам сборки. В манифесте сборки определены ограничения системы безопасности.

Ресурсы представляют собой, например, файлы изображений, помещаемых на форму, текстовые строки, значки приложения и т. д. Хранение ресурсов внутри сборки обеспечивает их защиту и упрощает развертывание приложения. Среда Visual Studio. NET предоставляет возможности автоматического внедрения ресурсов в сборку.

Открытые и частные сборки различаются по способам размещения на компьютере пользователя, именованию и политике версий. Частные сборки должны находиться в каталоге приложения, использующего сборку, или в его подкаталогах.

Открытые сборки размешаются в специальном каталоге, который называется глобальным кэшем сборок (Global Assembly Cache, GAC). Для идентификации открытой сборки используется уже упоминавшееся сильное имя (strong name), которое, как напоминает нам Троелсен, должно быть уникальным.

5.2 Библиотеки

Для создания библиотеки следует при разработке проекта в среде Visual Studio. NET выбрать шаблон Class Library (библиотека классов).

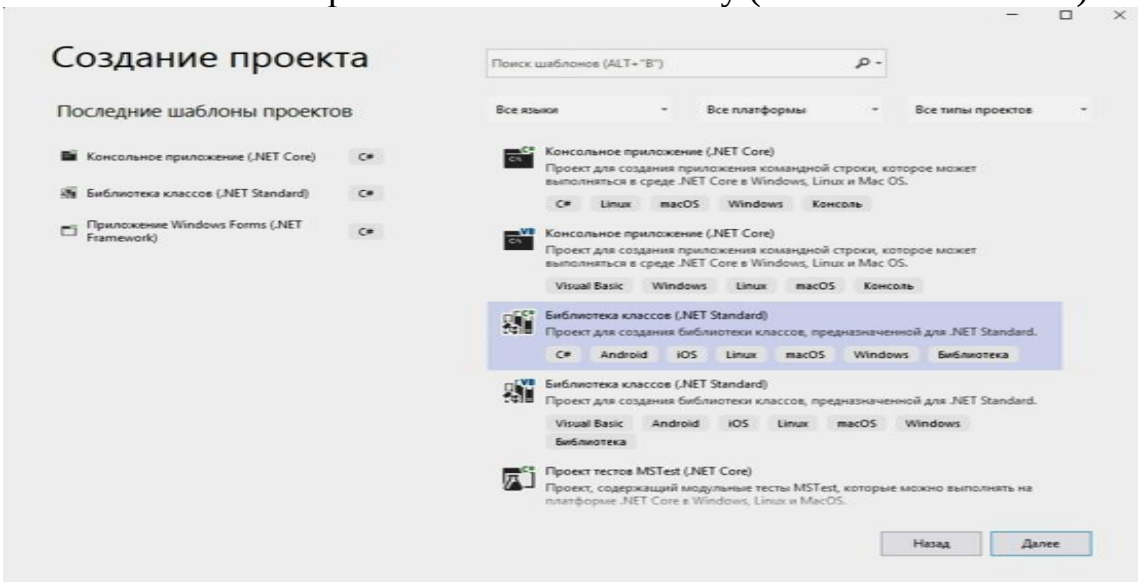

Рис. 37. Создание библиотеки

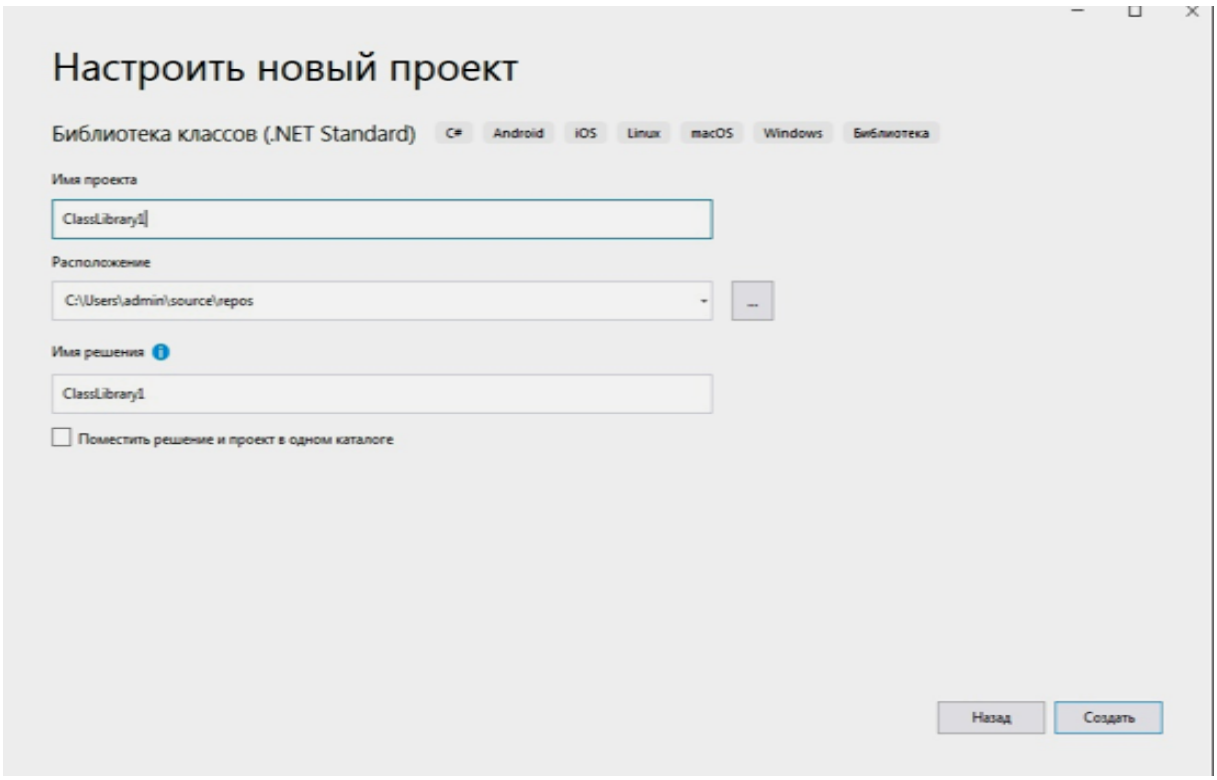

Рис. 38. Ввод имени библиотеки и её расположения

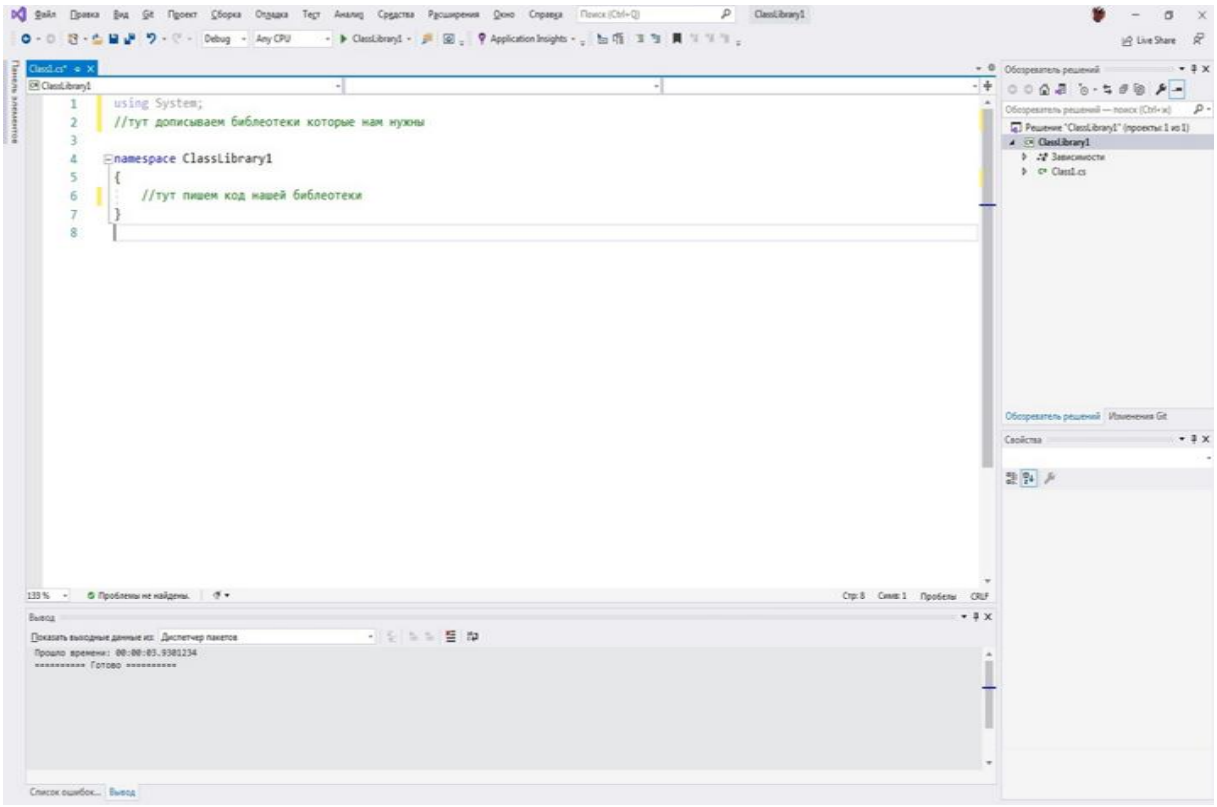

Рис. 39. Библиотека создана

Не следует забывать открывать доступ к нужным классам библиотеки с помощью спецификатора *public*.

Пример библиотеки учёта животных приведен ниже.

```
namespace CreatureLib
//Библиотека

{
   using System;

   public class Creature
   {
      public Creature()
      {
         this.hunger = true;
         this.name = "Noname";
      }
      public Creature(string name)
      : this()
      {
         this.name = name;
      }
      public Creature(string name, bool hunger)
      {
         this.hunger = hunger;
         this.name = name;
      }
      public bool Hunger
      {
         get
         {
            return hunger;
         }
         set
         {
            if (value == false)
               hunger = value;
            else
               hunger = true;
         }
      }
```

```
        public string Name
        {
            get
            {
                return name;
            }
        }
        public void Data()
        {
            Console.WriteLine($"Creature {name}   \tHungry = {hunger}");
        }
        string name; bool hunger;
    }
}
```

5.3 Использование библиотеки

Чтобы в своем проекте можно было применять созданную библиотеку, необходимо сначала произвести ее сборку (как это сделать было описано выше), а затем в нужный проект добавить ссылку на нее: перейти в *«Обозреватель решений»*, нажать правую кнопку мыши по разделу *«Ссылки»* и выбрать *«Добавить ссылку»*.

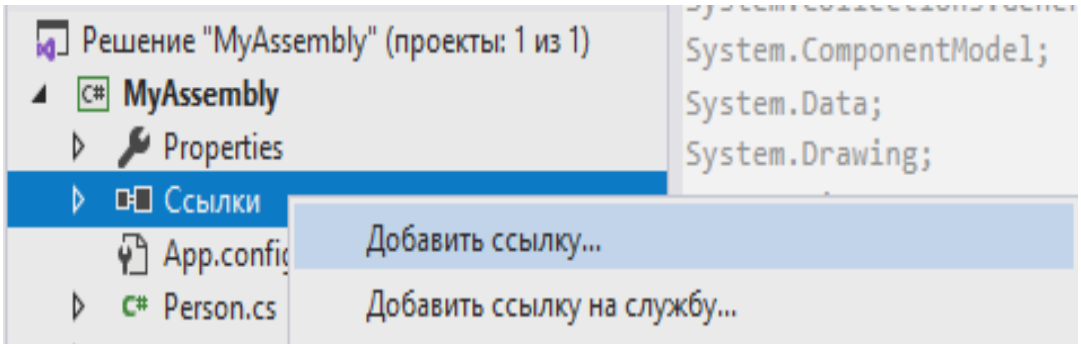

Рис. 40. Добавление ссылки на библиотеку

Далее в открывшемся окне необходимо кликнуть *«Обзор»* и выбрать нужный .dll файл библиотеки.

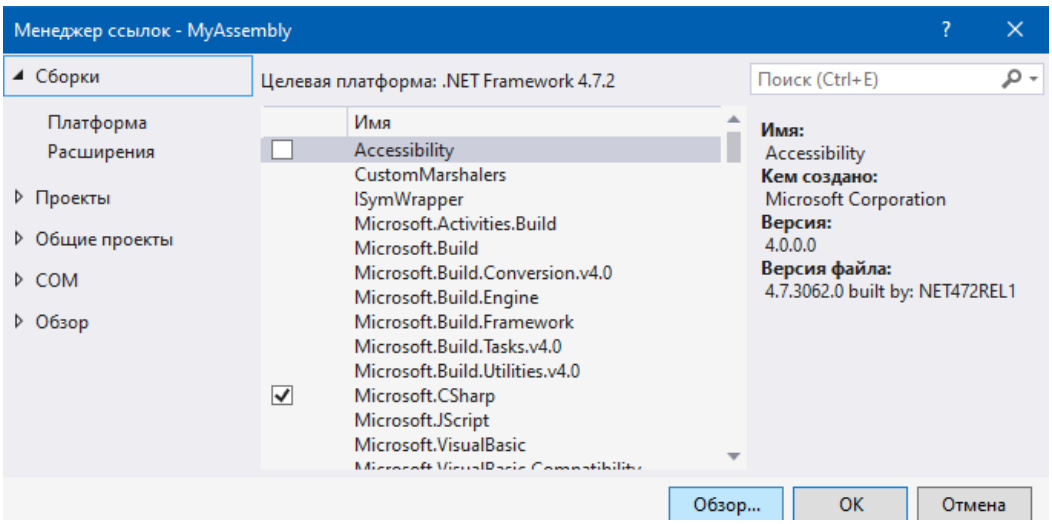

Рис. 41. Менеджер ссылок

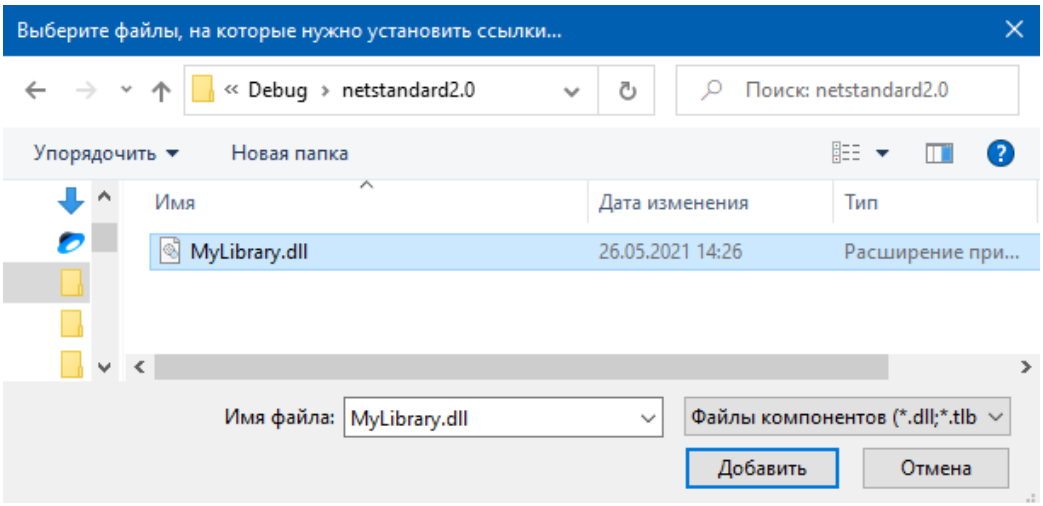

Рис. 42. Выбор библиотеки

После подключения библиотеки можно пользоваться ее открытыми элементами таким же образом, как если бы они были описаны в том же модуле.

Любая библиотека – это сервер, предоставляющий свои ресурсы клиентам. Создадим клиентское приложение, которое будет следить за сытостью животных в зоопарке, с использованием библиотеки CreatureLib.

Текст приложения приведен ниже.

```
using System;
using CreatureLib;

namespace ConsoleApp
```

```
{
   class Program
   {
      static void Main(string[] args)
      {
         const int n = 3;
         Creature[] name = new Creature[n];
         name[0] = new Creature("Lion", true);
         name[1] = new Creature("Tiger", false);
         name[2] = new Creature("Leopard", true);

         foreach (Creature x in name) x.Data();
         Console.Write("\n\nПокормить всех существ:");
         for (int i = 0; i < n; i++) name[i].Hunger = false;
         Console.WriteLine();
         foreach (Creature x in name) x.Data();
      }
   }
}
```

Преимущество .NET состоит в том, что благодаря стандартным соглашениям можно использовать библиотеки независимо от языка, на котором они были написаны. Таким образом, можно было бы написать клиентское приложение, например на языке VB.NET.

5.4 Атрибуты

Атрибуты – это дополнительные сведения об элементах программы (классах, методах, параметрах и т. д.). С помощью атрибутов можно добавлять информацию в метаданные сборки и затем извлекать ее во время выполнения программы. Атрибут является специальным видом класса и происходит от базового класса System.Attribute. Атрибуты делятся на стандартные и пользовательские. В библиотеке .NET предусмотрено множество стандартных атрибутов, которые можно использовать в программах. Если всего разнообразия стандартных атрибутов не хватит, чтобы удовлетворить прихотливые требования программиста, он может описать собственные классы атрибутов, после чего применять их точно так же, как стандартные.

При использовании (спецификации) атрибутов они задаются в секции атрибутов, для описания которого они предназначены. Секция заключается в квадратные скобки и может содержать несколько атрибутов,

перечисляемых через запятую. Порядок следования атрибутов произвольный.

Для каждого атрибута задаются имя, а также необязательные параметры и тип элемента сборки, к которому относится атрибут. Павловская приводит простейший пример атрибута:

```
[Serializable]
Class Example
{
        …
        [NonSerialized]
        string name;
        int year;
}
```

Атрибут [Serializable], означающий, что объекты этого класса можно сохранять во внешней памяти, относится ко всему классу Example. При этом поле name помечено атрибутом [NonSerialized], что говорит о том, что это поле сохраняться не должно.

Пример создания собственного атрибута:

```
using System;
using System.Windows.Forms;

namespace ExampleAttributes
{
   // Собственный атрибут для хранения информации о программе.
   class ProgramInfoAttribute : System.Attribute
   {
      public string Author { get; set; }
      public string Version { get; set; }

      public ProgramInfoAttribute(string author, string version)
      {
         Author = author;
         Version = version;
      }

      // Переопределение ToString для удобного отображения информации.
```

```
        public override string ToString()
        {
            return $"Автор: {Author}\nВерсия: {Version}";
        }
    }

    // Добавление атрибута, отражающего информацию о программе.
    [ProgramInfo("AuthorName", "1.0")]
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }

    public partial class Form1: Form
    {
        public string FirstName { get; set; }

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            // Получение типа класса, в котором содержится нужный
атрибут.
            Type type = typeof(Program);
            // Получение всех атрибутов класса.
            object[] attributes = type.GetCustomAttributes(false);

            foreach (Attribute attribute in attributes)
                // Находим нужный нам атрибут и показываем информацию
пользователю.
                if (attribute is ProgramInfoAttribute)
                {
```

```
                MessageBox.Show(attribute.ToString());
                break;
            }
        }
    }

    …
}
```

Результат работы программы можно увидеть на рисунке ниже.

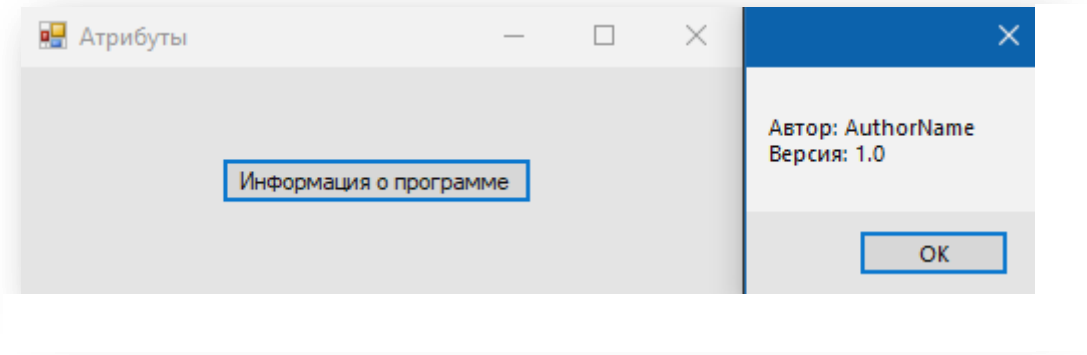

Рис. 43. Отображение информации о программе, хранящейся в атрибуте

Обычно из контекста понятно, к какому элементу сборки относится атрибут, однако в некоторых случаях могут возникнуть неоднозначности. Для их устранения перед именем атрибута записывается тип элемента сборки – уточняющее ключевое слово, отделяемое от атрибута двоеточием. Ключевые слова и соответствующие элементы сборки, к которым могут относиться атрибуты, перечислены в таблице.

Т а б л и ц а 3

Типы элемента сборки, задаваемые для атрибутов

Ключевое слово	**Описание**
assembly	Атрибут относится ко всей сборке
field	Атрибут относится к полю
event	Атрибут относится к событию
method	Атрибут относится к методу

Окончание табл. 3

Ключевое слово	Описание
param	Атрибут относится к параметрам метода
property	Атрибут относится к свойству
return	Атрибут относится к возвращаемому значению
type	Атрибут относится к классу или структуре

Пусть, например, перед методом описан гипотетический атрибут ABC:

```
[ABC]
public void Do()  {…}
```

По умолчанию он относится к методу. Чтобы указать, что атрибут относится не к методу, а к его возвращаемому значению, нужно написать:

```
[return:ABC]
public void Do()  {…}
```

Атрибут может иметь параметры. Они записываются в круглых скобках через запятую после имени атрибута и бывают позиционными и именованными. Именованный параметр указывается в форме имя = значение, для позиционного просто задается значение. Например, для использованного в следующем фрагменте кода атрибута CLSCompliant задан позиционный параметр true. Атрибуты, относящиеся к сборке, должны располагаться непосредственно после директив using, например:

```
using System;
[assembly: CLSCompliant(true)]
namespace ConsoleAplication1
{ …
```

Атрибут [CLSCompliant] определяет, удовлетворяет программный код соглашениям CLS (Common Language Specification) или нет.

Стандартные атрибуты, как и другие типы классов, имеют набор конструкторов, которые определяют, каким образом использовать (специфицировать) атрибут. Фактически при использовании атрибута

указывается наиболее подходящий конструктор, а величины, не указанные в конструкторе, задаются через именованные параметры в конце списка параметров. Стандартный атрибут [STAThread] относится к методу, перед которым он записан. Он имеет значение только для приложений, использующих модель COM, и задает модель потоков в рамках модели COM.

Атрибуты уровня сборки хранятся в файле AssemblyInfo.cs, автоматически создаваемом средой для любого проекта. Для явного задания номера версии сборки можно записать атрибут [AssemblyVersion], например

[assembly: AssemblyVersion(“1.0.0.0”)]

Еще несколько полезных стандартных атрибутов:

– *DefaultValue* – атрибут, переопределяющий начальное значение у свойств.

```
[DefaultValue("Неизвестно")]
public string FirstName { get; set; }
```

– Obsolete – помечает элемент программы как «устаревший», больше не актуальный.

```
[Obsolete]
public void ObsoleteMethod() { … }
```

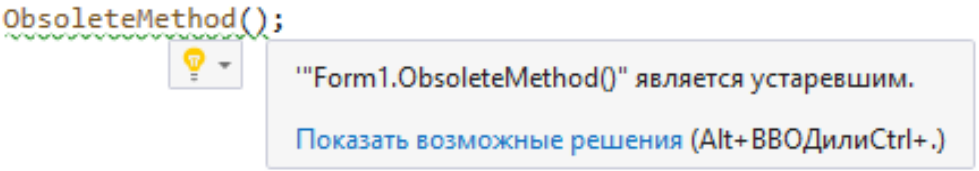

Рис. 44. Сообщение о том, что метод устарел

– DebuggerDisplay – позволяет быстро просматривать изменяемые данные во время отладки программы.

```
[DebuggerDisplay("firstName={firstName}, lastName={lastName}")]

class Person
{
   public string firstName = "John";
```

```
    public string lastName = "Smith";
}
```

Рис. 45. Компактное отображение значений полей класса во время отладки

Контрольные вопросы

1. Дать определение сборки.
2. Перечислить составные части сборки.
3. Что содержит манифест сборки?
4. Дать определение библиотеке.
5. Как создать библиотеку?
6. Как использовать библиотеку?
7. Дать определение атрибуту.
8. Как объявить атрибут?
9. Перечислить типы элемента сборки.
10. Как явно задать номер версии сборки?

Библиографический список

1. Павловская Т.А. C#. Программирование на языке высокого уровня: учебник для вузов. – СПб.: Питер, 2009. – 432 с.
2. Троелсен Э. C# и платформа .NET. Библиотека программиста. – СПб.: Питер, 2002. – 796 с.
3. Шилдт Г. C#: Учебный курс. – СПб.: Питер, 2002. – 512 с.
4. Шилдт Г. Полный справочник по C#. – М.: Вильямс, 2004. – 752 с.

Учебное издание

Мурлин Алексей Георгиевич
Мурлина Владислава Анатольевна
Янаева Марина Викторовна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Редактор Т.П. Горшкова
Компьютерная верстка В.А. Мурлина

Подписано в печать 30.09.2021 г. Формат 60x84/16
Бумага офсетная Офсетная печать
Печ. л. 9,75 Изд. № 138
Усл. печ. л. 9,1 Тираж 100 экз.
Уч.-изд. л. 6,9 Заказ № 395

ФГБОУ ВО «Кубанский государственный технологический университет»
350072, г. Краснодар, ул. Московская, 2, кор. А
Типография ФГБОУ ВО «КубГТУ»: 350058, г. Краснодар,
ул. Старокубанская, 88/4