

Министерство сельского хозяйства Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение
высшего образования
«Пермская государственная сельскохозяйственная академия
имени академика Д.Н. Прянишникова»

А.Ю. БЕЛЯКОВ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Пермь
ИПЦ «Прокростъ»
2017

УДК 004.42
ББК 32.973
Б 448

Рецензенты:

Рыбаков А.П. – профессор кафедры общей физики ПНИПУ, доктор физико-математических наук, профессор.

Шабуров А.С. – доцент кафедры автоматики и телемеханики ПНИПУ, кандидат технических наук, доцент.

Б 448 Беляков, А. Ю. Объектно-ориентированное программирование : учебное пособие / А.Ю. Беляков; М-во с.-х. РФ; федеральное гос. бюджетное образов. учреждение высшего образования «Пермская гос. с.-х. акад. им. акад. Д.Н. Прянишникова». – Пермь : ИПЦ «Прокрость», 2017. – 88 с.
ISBN 978-5-94279-358-6

Данное издание является базовым пособием по изучению основ парадигмы объектно-ориентированного программирования. В пособии в краткой форме изложены принципы объектно-ориентированного программирования и на практических примерах проанализированы некоторые приемы с использованием языка С#. Пособие ориентировано на самостоятельное освоение материала с исследованием программ в среде программирования Visual Studio. Рассматриваемый материал требует первичного знания основ структурного и событийного программирования.

Пособие предназначено для студентов, обучающихся по направлению подготовки 09.04.03 Прикладная информатика.

**УДК 004.42
ББК 32.973**

Утверждено в качестве учебного пособия Методическим советом ФГБОУ ВО Пермская ГСХА (протокол № 1 от 11 сентября 2017 г.).

Учебное издание

Беляков Андрей Юрьевич

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Подписано в печать 21.09.17.

Формат 60×84¹/₁₆. Усл. печ. л.5,5

Тираж 50 экз. Заказ № 117

ИПЦ «Прокрость»

Пермской государственной сельскохозяйственной академии
имени академика Д.Н. Прянишникова,

614990, Россия, г. Пермь, ул. Петропавловская, 23
тел. (342) 210-35-34

ISBN 978-5-94279-358-6

© ИПЦ «Прокрость», 2017
© Беляков А.Ю., 2017

Содержание

Введение.	4
Глава 1. Перегрузка функций.....	6
Глава 2. Структуры	9
Глава 3. Классы и объекты.....	12
Глава 4. Конструктор.....	21
Глава 5. Абстрагирование и инкапсуляция	23
Глава 6. Наследование	26
Глава 7. Полиморфизм	30
Глава 8. Интерфейсы	34
Глава 9. Делегаты и события.....	38
Глава 10. Библиотеки классов.....	54
Глава 11. Практика разработки библиотек классов	64
Глава 12. Библиотеки классов сторонних производителей	77
Задания для самостоятельного исполнения по темам.....	82
Заключение.....	84
Библиографический список.....	88

Введение

Любое программирование – это работа с абстракциями, реализующими математические, физические, экономические и иные модели процессов реального мира. Объектно-ориентированное программирование (ООП) – это программирование, ориентированное на использование объектов в коде программы. Порог вхождения начинающего программиста в парадигму ООП несколько выше, чем в структурное или событийное программирование. В структурном программировании программист использует прямой путь – как понимает алгоритм обработки данных (как правило, плоский линейный поток с добавлением структур ветвления и цикла), так и пишет его на языке программирования. Для вхождения в структурное программирование достаточно только освоить базовые алгоритмические конструкции и синтаксис операторов их исполняющих. В событийном программировании добавляется понимание разбиения всей программы на подпрограммы, работающие независимо и инициализирующиеся при возникновении событий, что характерно для таких операционных систем как, например, Windows или Android. Но решаемые компьютером задачи постоянно усложняются, и постепенно стало очевидным, что функционал языков программирования уже недостаточен.

Компьютерные программы, в целом, решают задачи автоматизации обработки данных, данных о реальном мире. Мир состоит из объектов, каждый из которых обладает некоторым определенным набором характеристик и возможных действий (функционала), то есть принадлежит к определенному классу. Именно этого принципа и придерживаются в процессе объектно-ориентированного программирования, где сначала описывают классы моделируемых сущностей, а затем

создают на их основе объекты. Класс содержит описание некоторой сущности в виде фиксированного количества её характеристик (поля) и допустимого функционала (методы). Объект создаётся по образу и подобию класса, содержит все описанные в классе методы и поля, с их конкретной реализацией. В качестве примера можно привести класс для сущности человек, с полями: Фамилия, Имя и Отчество и методом – сообщать своё Имя, и, созданный на основе данного класса объект с конкретными значениями полей: Иванов Иван Иванович, при обращении к единственному методу, данный объект вернет значение «Иван».

Разрабатывают классы и используют объекты на них основанные, потому что это удобно. Такой подход позволяет держать всю совокупность данных, описывающих ту или иную сущность, включая и различные функции, работающие с сущностью, в «одной корзине». Разработанный класс используют как новый тип данных, на основе которого в программе создают переменные данного типа и затем, через переменную-объект, пользуются полями и методами, заранее описанными в классе.

Глава 1. Перегрузка функций

Перегрузка функций – это способ многовариантной реализации функции в зависимости от типа данных аргументов или от их количества. Перегрузка функций присутствует не во всех языках программирования и, прежде всего, нужна для удобства кодирования и минимизации количества сущностей.

Структура минимальной программы с точкой входа:

```
using System;

class Program
{
    // тут размещаем свои функции
    static void Main(string[] args)
    {
        // тут размещаем вызовы функций
        Console.ReadKey();
    }
}
```

Добавим в программу несколько методов – перегружаемые функции:

```
static int sum(int x)
{
    return ++x;
}
static int sum(int x, int y)
{
    return x+=y;
}
static void inc(ref int x)
{
    x++;
}
static void inc(ref int x, int y)
{
    x+=y;
}
```

Возможные варианты обращений к функциям и результаты работы:

```
int x = 2, y = 3;
Console.WriteLine(sum(x)); // вывод 3
Console.WriteLine(sum(x,y)); // вывод 5
inc(ref x); Console.WriteLine(x); // вывод 3
inc(ref x, y); Console.WriteLine(x); // вывод 6
```

Если аргумент метода предварен ключевым словом `ref` (от слова *reference* – ссылка), то говорят, что аргумент передается в функцию не по значению, а по ссылке. В этом случае в оперативной памяти для переменной в основной программе и для аргумента функции не отводятся две разных области памяти, а как раз наоборот. Ссылка на одну и ту же область памяти позволяет через аргумент передавать значение в функцию и возвращать в основную программу. Перед вызовом функции с `ref` аргументом следует инициализировать значение соответствующей переменной:

```
int e=0; inc(ref e);
```

Есть альтернативный метод возвращения значения из функции: если аргумент метода предварить ключевым словом `out`, то соответствующую переменную можно не инициализировать начальным значением.

Пример функции и её вызов:

```
static void rnd(out int x)
{
    Random r = new Random();
    x = 1+r.Next(10);
}

int e; rnd(out e);
Console.WriteLine(e);
```

Рассмотренные методы аналогично используются и для передачи массивов.

Пример функции подсчета суммы элементов массива и её вызов:

```

static void sum_arr(int[] arr, out int sum)
{
    sum = 0;
    for(int i=0; i<arr.Length; i++)
    {
        sum += arr[i];
    }
}

int[] x = { 0, 1, 2, 3, 4 };
int s;
sum_arr(x, out s);
Console.WriteLine(s);

```

Примеры перегружаемых функций генерации элементов массива:

```

static void rnd_arr(ref int[] arr)
{
    Random r = new Random();
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = 1 + r.Next(10);
    }
}

static void rnd_arr(ref int[] arr, int max)
{
    Random r = new Random();
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = 1 + r.Next(max);
    }
}

static void rnd_arr(ref int[] arr, int max, int len)
{
    Random r = new Random();
    Array.Resize(ref arr, len);
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = 1 + r.Next(max);
    }
}

```


Варианты обращения к функции:

1:

```
int len = 5;
int[] arr = new int[len];
rnd_arr(ref arr, 3);
    foreach (int i in arr)
        Console.WriteLine(i);
```

2:

```
int[] arr = new int[0];
rnd_arr(ref arr, 2, 5);
foreach (int i in arr)
    Console.WriteLine(i);
```

Глава 2. Структуры

Структуры и классы в языке C# близкие понятия, так как обе сущности могут содержать и поля и методы работы. Основные отличия структуры от класса: невозможность использования конструктора без параметров, отсутствие наследования у структур, но, при этом, непосредственная адресация к структурам (не по ссылке как у классов) со всеми вытекающими из этого обстоятельства последствиями (особенности присваивания, *большая* скорость работы, экономия памяти). Исходя из этого, здесь мы кратко рассмотрим пример, демонстрирующий разницу в адресации структур и классов, и сразу после этого перейдем к исследованию классов.

```
struct posXY
{
    public int x, y;
}
class _posXY
{
    public int x, y;
}
```

```

posXY dot0 = new posXY();
posXY dot1 = new posXY();

dot0.x = 2; dot0.y = 4;
dot1.x = 5; dot1.y = 7;
dot1=dot0;
dot0.x = 6; dot0.y = 8;
Console.WriteLine(dot0.x + " " + dot0.y);
Console.WriteLine(dot1.x + " " + dot1.y);

_posXY _dot0 = new _posXY();
_posXY _dot1 = new _posXY();

_dot0.x = 2; _dot0.y = 4;
_dot1.x = 5; _dot1.y = 7;
_dot1 = _dot0;
_dot0.x = 6; _dot0.y = 8;
Console.WriteLine(_dot0.x + " " + _dot0.y);
Console.WriteLine(_dot1.x + " " + _dot1.y);

```

Вывод на экран:

```

6 8
2 4
6 8
6 8

```

Обратите внимание на «Вывод на экран», первые две строчки относятся к работе со структурами, оставшиеся две – с классами. Функционал строчек программного кода абсолютно одинаков, однако результаты различны. Это касается особенностей присваивания структур и классов. Непосредственная адресация к структурам приводит к тому, что при присваивании (`dot1=dot0;`) создается дубликат значения в оперативной памяти, как и при присваивании обычных переменных, поэтому значения первой и второй строчек различны. Адресация по ссылке при работе с классами приводит к обратному результату – при присваивании в новую переменную передается ссылка на объект класса, после чего обе пере-

менные ссылаются на один и тот же объект, поэтому третья и четвертая строчки совпадают.

Как правило, структуры используются для хранения и обработки групповых данных, в том числе, с данными разных типов (строки, целые и дробные числа, данные логического типа и т.п.). Рассмотрим краткий пример, демонстрирующий возможный вариант описания множества позиций шахматных фигур на доске с использованием массива фигур. Достаточное множество полей для описания фигуры можно задать следующим списком: Название фигуры (строковый тип), Цвет фигуры (символьный тип), координаты (целочисленный).

```
using System;

class Program
{
    struct posXY
    {
        public int x, y;
    }
    struct fig
    {
        public string name;
        public char color;
        public posXY pos;
    }
    static void Main(string[] args)
    {
        fig[] figs = new fig[2];
        figs[0].color = 'w';
        figs[0].name = "hourse";
        figs[0].pos.x = 1;
        figs[0].pos.y = 0;
        figs[1].color = 'w';
        figs[1].name = "king";
        figs[1].pos.x = 2;
        figs[1].pos.y = 2;
    }
}
```

Завершая обзор структур, и переходя к классам, ещё раз обращаю внимание читателя на важное обстоятельство – при присваивании структура ведет себя как обычная переменная, в то время как при работе с классами передается ссылка на объект соответствующего класса.

Глава 3. Классы и объекты

Прежде чем начать исследование классов определимся ещё раз с дефинициями: класс будем понимать как декларированную совокупность полей данных и функций для работы с ними, а объект как экземпляр класса.

При определении класса используют модификаторы доступа к полям и методам, в частности, `public`, `private`, `static`. Открытые поля и методы доступны из другого класса, скрытые – только из элементов класса, статические – общие для всех объектов данного класса.

Для большего понимания различия модификаторов доступа рассмотрим конкретный пример класса `Access`:

```
0  using System;
1
2  class Program
3  {
4      class Access
5      {
6          private byte x;
7          public const double gravity = 9.81;
8          public int count;
9          public int tempX
10         {
11             set
12             {
13                 x = Convert.ToByte(Math.Abs(value));
14                 count++;
            }
        }
    }
}
```

```

15         }
16         get { return x; }
17     }
18     public double result()
19     {
20         return Math.Sqrt(x);
21     }
22     public static string Name = "Ork";
23 }
24
25 static void Main(string[] args)
26 {
27     Access a = new Access();
28     a.tempX = -25;
29     Console.WriteLine(a.result());
30     a.tempX = 121;
31     Console.WriteLine(a.result());
32     Console.WriteLine(a.count);
33     Console.WriteLine(Access.Name);
34     Console.WriteLine(Access.gravity);
35     Console.ReadKey();
36 }
37 }

```

Вывод на экран:

```

5
11
2
Ork
9,81

```

Итак, в 27 строке мы создаем новый объект `a` на основе класса `Access`, в котором из переменных открыт доступ только к полям `tempX` и `count`. Основная переменная `x` – поле данного класса, с которой происходит работа, скрыта от пользователя. Безопасный доступ к переменной `x` реализован через аксессоры (сеттер – `set` и геттер – `get`) свойства `tempX` (строки 9-17).

Именно по этой причине принято разделять понятия поле и свойство. Свойством считают поле, «обернутое» в ак-

сессоры и, тем самым, получающее возможности контроля корректности ввода, вариации доступа к чтению и записи значения поля, привязки обработчиков событий к изменению значения поля.

В рассматриваемой программе с с помощью сеттера в строках 13 и 14 организована коррекция значения переменной до допустимо возможной величины и подсчет количества изменений данного поля. В строке 32 полученное значение выводится на экран.

Сделаем небольшое отступление по аксессорам, которые, как вы уже догадались, получили своё название от английского слова *access* (доступ). Дело в том, что уровень доступа к геттеру и сеттеру можно настроить индивидуально. В объектно-ориентированном программировании многие темы плотно взаимосвязаны, что затрудняет строго последовательное изложение материала с анализом усложняющихся примеров. В вопросе про аксессоры мы, немного забегаю вперед, коснемся дополнительного вопроса, а именно, наследования классов. В случае, если на этом этапе не все детали изложения будут понятны, имеет смысл дополнительно вернуться к данной теме впоследствии. Итак, для концентрации внимания и простоты изложения для начала сократим приведенный ранее пример до такого кода:

```
using System;

class Program
{
    class Access
    {
        private byte x;
        public int count;
        public int tempX
        {
            set
            {
```

```

        x = Convert.ToByte(Math.Abs(value));
        count++;
    }
    get { return x; }
}

static void Main(string[] args)
{
    Access a = new Access();
    a.tempX = -25;
    Console.WriteLine(a.tempX);
    a.tempX = 121;
    Console.WriteLine(a.tempX);
    Console.WriteLine(a.count);
    Console.ReadKey();
}
}

```

После запуска данной программы в консоли получим результат:

```

25
121
2

```

Мы видим, что сеттер работает и корректирует значение соответствующего поля. В завершающей строке значение 2 обозначает количество обращений к полю. С помощью модификаторов доступа мы можем сделать разным уровень доступа к геттеру и сеттеру, например, закрыть прямой доступ к сеттеру, оставив доступ к геттеру открытым:

```

using System;

class Program
{
    class Access
    {
        private byte x;
        public int count;
        public int tempX
        {
            protected set
            {

```

```

        x = Convert.ToByte(Math.Abs(value));
        count++;
    }
    get { return x; }
}
}
class Child : Access
{
    public void setX(int x)
    {
        this.tempX = x;
    }
}
static void Main(string[] args)
{
    Child a = new Child();
    a.setX(-25);
    Console.WriteLine(a.tempX);
    a.setX(121);
    Console.WriteLine(a.tempX);
    Console.WriteLine(a.count);
    Console.ReadKey();
}
}

```

Обратите внимание, что перед сеттером стоит модификатор `protected`, что означает, что данная опция доступна только из класса наследника. В данной программе класс `Access` является базовым, а класс `Child` – его наследником. Это означает, что объекту «а», созданному на основе класса `Child` будут доступны не только все свойства (поля) и методы `Child`, но и вся структура базового класса, включая и опции, помеченные как `protected`.

Класс `Child` обеспечивает доступ к свойству `tempX` через метод `setX`, но это не является обязательным или единственно возможным, вы вполне можете обойтись и созданием свойства (вместо метода `setX`):

```

class Child : Access
{
    public int setX
    {

```



```

        set { tempX = value; }
        get { return this.tempX; }
    }
    /*public void setX(int x)
    {
        this.tempX = x;
    }*/
}

```

Соответственно и реализация основной функции Main станет более очевидной:

```

static void Main(string[] args)
{
    Child a = new Child();
    a.setX = -25;
    Console.WriteLine(a.tempX);
    a.setX = 121;
    Console.WriteLine(a.tempX);
    Console.WriteLine(a.count);
    Console.ReadKey();
}

```

Заканчивая обсуждение аксессоров отдельно выделим причины и достоинства сеттеров и геттеров.

- осуществление контроля за корректностью значений полей;
- автоматизация коррекции вводимых данных;
- возможность сделать поле доступным только для чтения или только для записи;
- возможность организации дополнительных операций перед работой с полем, например, если обновление значения данного поля должно вызывать некоторое действие как в событийном программировании.

Пора вернуться к анализу обсуждаемого примера класса Access. В представленном классе используется константа (строка 7), обращение к ней возможно только через имя класса

(строка 34). Аналогичным образом обращение к статическому полю (строка 22) возможно только через имя класса (строка 33). Работа с методами класса организуется аналогичным образом с использованием модификаторов доступа.

По умолчанию элементы класса считаются закрытыми, это означает, что вместо такого объявления поля:

```
private byte x;
```

вполне позволительно написать сокращенный вариант:

```
byte x;
```

Заручившись полученными знаниями, разработаем класс с практическим применением. Это будет класс для шахматной фигуры «Конь», в котором будут предусмотрены:

- открытые поля (`public int posX, posY`) для задания координат фигуры вида «номер строки, номер столбца» в диапазоне от 0 до 7;

- скрытая функция (`private string pos_name(int row, int column)`) для приведения позиции на шахматной доске вида «номер строки, номер столбца» к строковому виду (например, «E2»);

- скрытая функция (`private int pos_full(int row, int column)`) для перевода координат вида «номер строки, номер столбца» к номеру позиции на шахматной доске (от 0 до 63);

- открытая функция (`public List<string> f1()`) для формирования списка полей, бьющихся «Конём»:

```
using System;
using System.Collections.Generic;

class Program
{
    class hourse
    {
```

```

    public int posX, posY;
    private int pos_full(int row, int column)
    {
        return column + row * 8;
    }
    private string pos_name(int row, int column)
    {
        const string ind = "ABCDEFGH";
        return ind.Substring(column, 1) + Convert.ToString(row+1);
    }
    public List<string> f1()
    {
        List<string> pos_arr = new List<string>();
        int _hr, _hc, hr, hc, _pos_h, pos_h;
        pos_h = pos_full(this.posY, this.posX);
        for (int r = 0; r < 8; r++)
            for (int c = 0; c < 8; c++)
            {
                _pos_h = pos_full(r, c);
                _hr = _pos_h / 8; _hc = _pos_h % 8;
                hr = pos_h / 8; hc = pos_h % 8;
                if ((Math.Abs(_hr - hr) == 1) && (Math.Abs(_hc - hc) == 2) ||
                    (Math.Abs(_hr - hr) == 2) && (Math.Abs(_hc - hc) == 1))
                    pos_arr.Add(pos_name(r, c));
            }
        return pos_arr;
    }
}

static void Main(string[] args)
{
    List<string> listPos = new List<string>();
    hourse h = new hourse();
    h.posX = 0; h.posY = 1;
    listPos=h.f1();
    foreach (string s in listPos)
        Console.WriteLine(s);
    Console.ReadKey();
}
}

```

Возможна и иная организация функции, формирующий список «битых» полей и возвращающей его не через своё имя, а через аргумент функции (аналог процедуры в Delphi):

```
public void f2(ref List<string> pos_arr)
{
    int _hr, _hc, hr, hc, _pos_h, pos_h;
    pos_h = pos_full(this.posY, this.posX);
    for (int r = 0; r < 8; r++)
        for (int c = 0; c < 8; c++)
        {
            _pos_h = pos_full(r, c);
            _hr = _pos_h / 8; _hc = _pos_h % 8;
            hr = pos_h / 8; hc = pos_h % 8;
            if ((Math.Abs(_hr - hr) == 1) && (Math.Abs(_hc - hc) == 2) ||
                (Math.Abs(_hr - hr) == 2) && (Math.Abs(_hc - hc) == 1))
                pos_arr.Add(pos_name(r, c));
        }
}
```

Обращение к такой функции, соответственно, происходит без непосредственного присваивания результата, а через передачу значения аргумента. Если вызов метода класса в первом варианте реализации выглядел так:

```
listPos=h.f1();
```

то в новом варианте реализации – так:

```
h.f2(ref listPos);
```

Принципиальной разницы между данными способами передачи значений функции нет. В первом случае удобнее передавать одиночное значение, во втором – когда из функции нужно вернуть несколько значений разных типов (но и это можно обойти созданием структуры).

Глава 4. Конструктор

Конструктор будем понимать как особую функцию для инициализации объекта. Инициализация объекта означает присвоение некоторым полям объекта начальных значений. Имя функции конструктора совпадает с именем класса. Конструкторов может быть несколько. Конструкторы могут отличаться количеством передаваемых в них параметров. Конструктор может не иметь параметров. Если в классе не определен ни один конструктор, то значения полей инициализируются нулем [1].

Рассмотрим варианты реализации конструктора на примере класса с методом, возвращающим вес предмета, если известна его масса. Получаемый вес будет вычисляться по формуле $weight = massa * gravity$, то есть через массу и ускорение свободного падения, которая отличается на Земле, Луне и иных космических телах. Так как чаще всего будет вычисляться вес именно на Земле, то создадим так называемый *конструктор по умолчанию*, который будет инициализировать поле `gravity` значением 9,81. Для иных случаев создадим конструктор, в который будет возможность передать значение ускорения свободного падения (в примере, приведенном ниже, расчет выполнен для Луны).

```
0 using System;
1
2 class Program
3 {
4     class Builder
5     {
6         public Builder()
7         {
8             this.gravity = 9.81;
9         }
```

```

10     public Builder(double gravity)
11     {
12         this.gravity = gravity;
13     }
14     double gravity;
15     public double weight(double massa)
16     {
17         return massa * this.gravity;
18     }
19 }
20 static void Main(string[] args)
21 {
22     double m = 5.0, moon = 1.62;
23
24     Builder a = new Builder();
25     Console.WriteLine(a.weight(m) + " НЬЮТОН");
26     Builder b = new Builder(moon);
27     Console.WriteLine(b.weight(m) + " НЬЮТОН");
28     Console.ReadKey();
29 }
30 }

```

В данной программе последовательно создаются два объекта а и b. В первом объекте поля инициализируются конструктором по умолчанию, а во втором объекте – конструктором с параметром.

Глава 5. Абстрагирование и инкапсуляция

Абстрагирование будем понимать как особый способ организации работы с данными класса. Основная идея стоит в упрощении и унификации работы с данными при сохранении надежности функционирования класса. Такой подход позволяет работать с объектами, не вдаваясь в особенности их реализации – абстрагироваться от несущественных на данном уровне программирования деталей. Абстрагирование реализуется такими частными концепциями как инкапсуляция, наследование и полиморфизм. Далее последовательно рассмотрим каждую из них.

Инкапсуляцию будем понимать как предоставление различных уровней доступа (вплоть до полного сокрытия) к полям и функциям класса [2]. Модификаторы `private` и `public` мы уже использовали ранее. Здесь имеет смысл рассмотреть ещё один модификатор `protected`. Мы немного пересечемся со следующей главой, но они взаимосвязаны и это вынужденная необходимость. Дело в том, что, если член класса с модификатором `private` доступен только методам внутри класса, а член класса с модификатором `public` доступен и «внутри» класса и «снаружи», то вот член модификатор `protected` имеет смысл только при наследовании. Член базового класса с модификатором `protected` доступен как «внутри» класса так и в производном классе, но не в других классах. Для полного понимания рассмотрим простейший пример:

```
0 using System;
1
2 class Program
3 {
4     class Base
5     {
```

```

6         public int a = 1;
7         protected int b = 2;
8         private int c = 3;
9         protected int sum()
10        {
11            return b + c;
12        }
13    }
14    class Save: Base
15    {
16        public int x = 4;
17        private int y = 5;
18        public int _b
19        {
20            set { this.b = value; }
21            get { return this.b; }
22        }
23        public int _sum()
24        {
25            return this.sum();
26        }
27    }
28    static void Main(string[] args)
29    {
30        Base obj1 = new Base();
31        Save obj2 = new Save();
32
33        Console.WriteLine(obj1.a);
34        //Console.WriteLine(obj1.b);
35        //Console.WriteLine(obj1.c);
36        Console.WriteLine();
37        Console.WriteLine(obj2.a);
38        Console.WriteLine(obj2._b);
39        Console.WriteLine(obj2._sum());
40        Console.WriteLine(obj2.x);
41        //Console.WriteLine(obj2.y);
42        Console.ReadKey();
43    }
44 }

```


В приведенном примере введены два класса: базовый и производный от него, наследующий его поля и методы. В строках 30 и 31, на основе этих классов, создаются два соответствующих объекта. Далее (с 33 строки) производится попытка вывода в консоль значений полей объектов и возвращаемых функциями значений. Те из них, которые не могут быть использованы и, на которые компилятор сгенерирует ошибку, взяты в комментарий. Так как объект `obj1` наследует поля и методы класса `Base`, то к выводу доступно только одно поле с модификатором `public`, остальные имеют более высокий уровень закрытости `protected` и `private`. Однако объект `obj2` наследует поля и методы и родного класса `Save` и родительского класса `Base`. Сам класс `Save` имеет доступ к полям и методам класса `Base` с модификаторами `public` и `protected`, поэтому посредством методов класса `Save` (строки 18–26) к выводу становятся доступны (строки 38, 39) не только поля и методы `public` класса `Base`, но и имеющие более высокий уровень закрытия (`protected`).

Глава 6. Наследование

Наследование будем понимать как процесс создания новых (производных) классов из уже существующих [3]. Производный класс получает все возможности базового класса, но может также быть дополнен своими полями и методами. От одного базового класса может быть несколько «наследников», в свою очередь, от каждого из них тоже могут быть наследуемые классы. Приведем простейший пример наследования: пусть необходимо создать классы субъектов компьютерной игры: воины, крестьяне, орки. Для такого подмножества можно разработать следующую иерархию классов:

- базовый класс «живое»;
- от него наследуемые классы «человек» и «орк»;
- от класса «человек» наследники «воин» и «крестьянин».

Каждый из классов нужно снабдить необходимым (минимальным и достаточным) функционалом (полями и методами). Обычно в таких случаях рисуют диаграмму классов UML для наглядного отображения иерархии классов, полей, методов и модификаторов доступа к ним. В нашем случае можно обойтись непосредственно описанием.

В базовый класс поместим поле «возраст», так как это характерно для всего живого. Орку добавим характеристику «Цвет» двух вариантов – белый и черный, что влияет на способ взаимодействия с человеком (в данной программе геймплей не проработан для обеспечения компактности материала). Цвет орка выбирается случайным образом в конструкторе.

Человеку добавим пол и имя. Воину добавим уровень вооружения в диапазоне от 0 до 1, при рождении дается 1.

Крестьянину добавим уровень обороны в диапазоне от 0 до 1, при рождении дается 1.

Ниже приведена программа, реализующая описанные классы, и демонстрирующая пример создания на их основе объектов.

```
0  using System;
1
2  class Program
3  {
4      class Animal
5      {
6          public int age;
7      }
8      class Human: Animal
9      {
10         public bool sex;
11         public string name;
12         public Human()
13         {
14             this.name = "Anonim";
15             this.sex = false;
16         }
17         public Human(char pol, string nme)
18         {
19             this.name = nme;
20             this.sex = pol=='m' ;
21         }
22         public Human(char pol, string _name, int _age)
23         {
24             this.name = _name;
25             this.sex = pol == 'm';
26             this.age = _age;
27         }
28     }
29     class Ork: Animal
30     {
31         private string[] цвета = { "White" , "Black" };
32         public string color;
33         public Ork()
```

```

34         {
35             Random r = new Random((int)DateTime.Now.Ticks);
36             this.color = цвета[r.Next(2)];
37         }
38     }
39     class Warrior: Human
40     {
41         public double weapon;
42         public Warrior()
43         {
44             this.weapon = 1.0;
45         }
46     }
47     class Farmer : Human
48     {
49         public double defense;
50         public Farmer()
51         {
52             this.defense = 1.0;
53         }
54     }
55     static void Main(string[] args)
56     {
57         Ork hero_1 = new Ork();
58         Console.WriteLine(hero_1.age);
59         Console.WriteLine(hero_1.color);
60
61         Ork hero_2 = new Ork();
62         Console.WriteLine(hero_2.age);
63         Console.WriteLine(hero_2.color);
64
65         Warrior hero_3 = new Warrior();
66         Console.WriteLine(hero_3.age);
67         Console.WriteLine(hero_3.weapon);
68
69         Farmer hero_4 = new Farmer();
70         Console.WriteLine(hero_4.age);
71         Console.WriteLine(hero_4.defense);
72
73         Human hero_5 = new Human();
74         Console.WriteLine(hero_5.age);
75         Console.WriteLine(hero_5.sex);

```

```

76         Console.WriteLine(hero_5.name);
77
78         Human hero_6 = new Human('m', "Иван", 24);
79         Console.WriteLine(hero_6.age);
80         Console.WriteLine(hero_6.sex);
81         Console.WriteLine(hero_6.name);
82
83         Console.ReadKey();
84     }
85 }

```

Возможный вариант вывода результатов данной программы:

```

0
Black
0
White
0
1
0
1
False
Anonim
24
True
Иван

```

Обратите внимание на вывод данных о последних двух героях игры – оба они относятся к классу «Человек», но при этом не являются «Воином» или «Крестьянином». Наглядно показано, что не только классы наследники могут самостоятельно создавать объекты и обладать несколькими вариантами конструкторов.

Глава 7. Полиморфизм

Полиморфизм будем понимать как интерфейс пользователя с реализацией, зависящей от типа объекта. Полиморфизм дает возможность объектам (разным «наследникам» базового класса) с одинаковой спецификацией иметь различную реализацию некоторых методов.

Таким образом, можно сказать, что полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с использованием процесса, который называется переопределением метода (*method overriding*).

Виртуальным называется такой метод, который объявляется как *virtual* в базовом классе. Виртуальный метод отличается тем, что он может быть переопределен в одном или нескольких производных классах. Следовательно, у каждого производного класса может быть свой вариант виртуального метода. Кроме того, виртуальные методы интересны тем, что именно происходит при их вызове по ссылке на базовый класс. В этом случае средствами языка C# определяется именно тот вариант виртуального метода, который следует вызывать, исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения. Поэтому при ссылке на разные типы объектов выполняются разные варианты виртуального метода.

Метод объявляется как виртуальный в базовом классе с помощью модификатора *virtual*, указываемого перед его именем. Когда же виртуальный метод переопределяется в производном классе, то для этого используется модификатор *override*. А сам процесс повторного определения виртуального метода в производном классе называется переопределением метода. При переопределении метода – имя, возвращаемый

тип и сигнатура переопределяющего метода должны быть точно такими же, как и у того виртуального метода, который переопределяется.

Переопределение метода служит основанием для воплощения одного из самых эффективных в С# принципов: динамической диспетчеризации методов [4], которая представляет собой механизм разрешения вызова во время выполнения, а не компиляции. Значение динамической диспетчеризации методов состоит в том, что именно благодаря ей в С# реализуется динамический полиморфизм. Если же, при наличии многоуровневой иерархии, виртуальный метод не переопределяется в производном классе, то выполняется ближайший его вариант, обнаруживаемый вверх по иерархии.

И еще одно замечание: свойства также подлежат модификации ключевым словом `virtual` и переопределению ключевым словом `override`.

В качестве примера рассмотрим программу со следующей иерархией классов:

- базовый класс «Животное» с полями «Место проживания» и «Название животного» с сеттерами и геттерами по умолчанию и виртуальным методом – функцией вывода в консоль места проживания;

- и его классы наследники «Кенгуру», «Слон» и «Кит» с конструкторами по умолчанию, задающими наименование и место проживания соответствующих животных. Обратите внимание, что в классе «Кит» не переопределяется метод `Print`.

```
0 using System;
1 using System.Collections.Generic;
2
3 class Program
4 {
```

```

5     class Animal
6     {
7         public string place { get; set; }
8         public string name { get; set; }
9         public const string frase = " place to live - ";
10        public virtual void Print()
11        {
12            Console.WriteLine("-----");
13            if (name!=null) Console.WriteLine("Animal name - " +
14 name);
15            Console.WriteLine("Animal place to live - " + place);
16            Console.WriteLine("-----");
17        }
18    }
19    class Kangaroo : Animal
20    {
21        public Kangaroo()
22        {
23            this.name = "Кенгуру";
24            this.place = "Австралия";
25        }
26        public override void Print()
27        {
28            Console.WriteLine(name + frase + place);
29        }
30    }
31    class Elephant : Animal
32    {
33        public Elephant()
34        {
35            this.name = "Слон";
36            this.place = "Африка";
37        }
38        public override void Print()
39        {
40            Console.WriteLine(name + frase + place);
41        }
42    }
43    class Whale : Animal
44    {
45        public Whale()
46        {

```



```

47         this.name = "Кит";
48         this.place = "Океан";
49     }
50 }
51 static void Main(string[] args)
52 {
53     List<Animal> animals = new List<Animal>();
54     animals.Add(new Animal());
55     animals.Add(new Elephant());
56     animals.Add(new Kangaroo());
57     animals.Add(new Whale());
58
59     animals[0].place = "Планета Земля";
60     animals[3].place = "Тихий океан";
61
62     foreach (Animal animal in animals)
63     {
64         animal.Print();
65     }
66
67     Console.ReadKey();
68 }
}

```

По результатам работы этой программы в консоль будут выведены следующие строки:

```

-----
Animal place to live - Планета Земля
-----
Слон place to live - Планета Земля
Кенгуру place to live - Планета Земля
-----
Animal name - Кит
Animal place to live - Тихий океан
-----

```

Обратите внимание на реализацию принципа динамического полиморфизма в случае выполнения метода Print для «Кита».

Глава 8. Интерфейсы

Интерфейс класса будем понимать как именованный набор свойств и сигнатур методов, без их конкретной реализации [5]. Сама реализация будет конкретизирована в каждом классе наследнике интерфейса индивидуально. Следует уточнить, что в C# от одного родительского класса может быть образовано несколько классов наследников, но не наоборот. Нельзя организовать иерархию классов так, чтобы у одного класса наследника было несколько родительских классов. Для решения этой проблемы можно использовать технологию интерфейсов, которая позволяет классу наследовать от нескольких интерфейсов.

Кроме того предварительное описание интерфейсов обязывает реализовать все заявленные в них методы, приводит к унификации классов наследников и не позволяет компилировать библиотеку классов с не полностью описанным классом наследником интерфейса.

Наследование интерфейсов разными классами очень похоже на полиморфизм методов в классах, но с одним важным дополнением. При полиморфизме объявленный в родительском классе виртуальный метод может получить частную реализацию в каждом из классов наследников. При наследовании интерфейсов объявленный метод не просто может, но и обязан получить частную реализацию в каждом из классов наследников. Удобство такого подхода состоит ещё и в том, что можно объекты разных классов объединять в один список объектов и затем в цикле обращаться к определенному методу интерфейса, который реализован в каждом классе по-своему. Ниже приведен пример интерфейса и двух классов-наследников, демонстрирующих данный подход.

```
using System;
using System.Collections.Generic;

class Program
{
    public interface IBase
    {
        int x { set; get; }
    }
}
```

```

        int getX();
    }

    class A : IBase
    {
        protected int _x;
        public int x
        {
            set { _x = value; }
            get { return _x / 10; }
        }
        public int getX()
        {
            return _x;
        }
    }

    class B : IBase
    {
        protected int _x;
        public int x
        {
            set { _x = value; }
            get { return _x % 10; }
        }
        public int getX()
        {
            return _x;
        }
    }

    static void Main(string[] args)
    {
        List<IBase> obs = new List<IBase>();
        obs.Add(new A());
        obs.Add(new B());
        Random r = new Random();
        foreach (IBase ob in obs)
        {
            ob.x = r.Next(10, 100);
            Console.WriteLine(ob.x + " " + ob.getX());
        }
        Console.ReadKey();
    }
}

```

В интерфейсе IBase объявлено свойство (поле x с аксессорами) и метод getX. В программе представлены два класса A и B, наследники интерфейса IBase. В основной программе сначала создается список объектов, затем в список

добавляются объекты разных классов (А и В). После чего циклом `foreach` перечисляются все объекты в списке и в каждом из них используются одинаковые свойства и методы, что обеспечивает общий интерфейс. Такое использование объектов разных типов в одном перечислении и является одним из весомых преимуществ наследования интерфейсов.

Коротко рассмотрим работу методов, основанных на интерфейсе `IBase`. В свойство `x` подаются случайные двузначные числа, но проходя через сеттеры классов А и В числа сохраняются в скрытом поле `_x`. В программе организован вывод данных, который позволяет проанализировать работу свойств и методов. Рассмотрим пример вывода:

```
3 32
9 29
```

В частности во время исполнения команды вывода в консоль

```
Console.WriteLine(ob.x + " " + ob.getX());
```

первым выводится значение свойства `x` у каждого объекта, а затем, через символ пробела, выводится скрытое поле `_x` методом `getX()`. Так как объекты разные, то и геттеры их работают по-разному, хотя обращение к ним абсолютно идентично – `ob.x`. Геттер объекта класса А выводит первую цифру двухзначного числа (3 для 32), а геттер объекта класса В – последнюю (9 для 29). Если же мы хотим получить само двухзначное число, то обращаемся к методу `getX()` так: `ob.getX()`.

Завершая рассмотрение интерфейсов, рассмотрим небольшой пример наследования нескольких интерфейсов одним классом:

```
using System;

class Program
{
    public interface IBase
    {
        int x { get; }
    }
    public interface IGetX
    {
        int getX();
    }
}
```

```

class A : IBase, IGetX
{
    protected int _x;
    public A()
    {
        _x = 0;
    }
    public A(int d)
    {
        _x = d;
    }
    public int x
    {
        get { return _x / 10; }
    }
    public int getX()
    {
        return _x;
    }
}

static void Main(string[] args)
{
    Random r = new Random();
    int count = 10;
    A[] arr = new A[count];
    for (int i=0; i<count; i++)
    {
        if (i%2>0)
            arr[i] = new A(r.Next(10, 100));
        else
            arr[i] = new A();
    }
    foreach (A ob in arr)
        Console.WriteLine(ob.x + " " + ob.getX());
    Console.ReadKey();
}

```

В данной программе используется класс A, основанный на двух интерфейсах IBase и IGetX. В начале программы создается массив объектов класса A. В классе реализованы два конструктора: по умолчанию и с параметром, инициализация которых зависит от четности индекса объекта в массиве arr. Остальные особенности программы вы сможете проанализировать сами, опираясь на предыдущий пример.

Глава 9. Делегаты и события

Делегатом называют особый тип данных, который может ссылаться на метод. Формат создания делегата подобен описанию интерфейса. Только речь идет не о возможности наследования в разных классах сигнатуры метода без описания его реализации, а о создании объекта делегата, содержащего ссылку на конкретный метод с описанной реализацией.

Под сигнатурой будем понимать шаблон метода, включающий тип возвращаемого функцией значения и типы данных аргументов, например:

- double (int, byte),
- void (string),
- bool (int).

Таким образом, объект делегата содержит не только адрес метода, но и аргументы (при необходимости) и возвращаемое значение. Самое важное заключается в том, что этот метод можно вызывать по ссылке, то есть именно делегат позволяет вызывать метод, на который он ссылается. Еще одно замечание о способностях делегатов состоит в их способности менять ссылку на конкретный метод во время выполнения программы, то есть динамически, а не только на этапе разработки. Данная способность предоставляет программисту право писать функции в библиотеке классов в обобщенном виде, делая классы универсальными в использовании для разных типов приложений (консольные, Windows Forms и др.).

Для лучшего понимания описанных преимуществ рассмотрим несколько примеров и начнем с реализации делегатов, ссылающихся на различные арифметические операции:

```
using System;

namespace ООП_delegate_1
{
    delegate int Operation(int i, int j);

    class Program
    {
        // методы с одинаковой сигнатурой
        static int summa(int x, int y)
```

```

    {
        return x + y;
    }
    static int subtract(int x, int y)
    {
        return x - y;
    }
    static int multiply(int x, int y)
    {
        return x * y;
    }

    static int divide(int x, int y)
    {
        return x / y;
    }

    static void Main()
    {
        int result;
        // создадим объект - делегат
        Operation op;

        // установим ссылку на метод
        op = new Operation(summa);
        result = op(2, 8);
        Console.WriteLine("Сумма: " + result);

        // Изменим ссылку на метод
        op = new Operation(subtract);
        result = op(2, 8);
        Console.WriteLine("Разность: " + result);

        Console.ReadLine();
    }
}

```

В данной программе сначала мы описываем делегат как тип, задавая сигнатуру методов и тип возвращаемого значения:

```
delegate int Operation(int i, int j);
```

По сути, мы обращаемся к делегату как к классу, именно по этой причине мы имеем право уже в самой программе объявить переменную типа, указанного в делегате:

```
Operation op;
```

и, в дальнейшем, создать объект делегата:

```
op = new Operation(summa);
```

для которого, при необходимости, динамически изменить ссылку на используемый метод:

```
op = new Operation(subtract);.
```

Таким образом, можно сделать вывод, что работа с делегатом осуществляется как с любым классом. Попробуем закрепить это понимание, дополнив нашу программу классом со статическими методами, к которым будут обращаться объекты – делегаты:

```
using System;

class Program
{
    // делегаты
    delegate int intOperat(int i, int j);
    delegate double dblOperat(double i, double j);

    class mathem
    {
        // методы с одинаковой сигнатурой
        public static int summa(int x, int y)
        {
            return x + y;
        }
        public static int subtract(int x, int y)
        {
            return x - y;
        }
        public static int multiply(int x, int y)
        {
            return x * y;
        }
        // методы с одинаковой сигнатурой
        public static double divide(double x, double y)
        {
            return x / y;
        }
        public static double power(double x, double y)
        {
            return Math.Pow(x, y);
        }
    }

    static void Main()
    {
        // создадим объекты - делегаты
        intOperat op1;
```



```

        dblOperat op2;

        // установим ссылку на метод
        op1 = new intOperat(mathem.summa);
        Console.WriteLine("Сумма: " + op1(2, 3));

        // изменим ссылку на метод
        op1 = new intOperat(mathem.subtract);
        Console.WriteLine("Разность: " + op1(2, 3));

        // установим ссылку на метод
        op2 = new dblOperat(mathem.divide);
        Console.WriteLine("Частное: " + op2(2, 3));

        // изменим ссылку на метод
        op2 = new dblOperat(mathem.power);
        Console.WriteLine("Степень: " + op2(2, 3));

        Console.ReadLine();
    }
}

```

В программу добавлены комментарии, которые помогут разобраться с кодом. Учитывая, что мы можем создать переменную типа, описанного в делегате, то и созданный на основе делегата объект можно передавать в качестве аргумента. Обратите внимание, что можно передавать объект-делегат, который имеет ссылку на конкретную реализацию метода с заданной сигнатурой, что дает возможность управлять работой программы объектами:

```

using System;

class Program
{
    delegate double dblOperat(double i, double j);

    public static double divide(double x, double y)
    {
        return x / y;
    }
    public static double power(double x, double y)
    {
        return Math.Pow(x, y);
    }

    static double StartDelegate(dblOperat oper, double a, double b)

```

```

    {
        return oper(a, b);
    }

    static void Main()
    {
        double x = 2, y = 3;
        Console.WriteLine("Частное: {0:F4}",
            StartDelegate(divide, x, y));
        Console.WriteLine("Степень: {0:F4}",
            StartDelegate(power, x, y));

        Console.ReadLine();
    }
}

```

Особое внимание обратите на эти строчки:

```

Console.WriteLine("Частное: {0:F4}", StartDelegate(divide, x, y));
Console.WriteLine("Степень: {0:F4}", StartDelegate(power, x, y));

```

Для реализации такого подхода мы создали специальную функцию `StartDelegate`, через аргументы которой мы можем выбрать необходимый метод и передать аргументы.

Обычно необходимые для реализации программы методы не раскидывают просто так по коду, а объединяют в класс. Перепишем нашу программу, объединив методы в класс, а в теле основной программы создадим на основе делегата соответствующие объекты и уже, пользуясь ими, будем выполнять вычислительную задачу:

```

using System;

class Program
{
    public delegate double dblOperat(double i, double j);

    public class Oprt
    {
        public double divide(double x, double y)
        {
            return x / y;
        }
        public double power(double x, double y)
        {
            return Math.Pow(x, y);
        }
    }
}

```

```

    }

    static double StartDelegate(dblOperat oper, double a, double b)
    {
        return oper(a, b);
    }

    static void Main()
    {
        Oprt op = new Oprt();
        double x = 2, y = 3;

        dblOperat del1 = op.divide;
        dblOperat del2 = op.power;

        Console.WriteLine("Частное: {0:F4}",
            StartDelegate(del1, x, y));
        Console.WriteLine("Степень: {0:F4}",
            StartDelegate(del2, x, y));

        Console.ReadLine();
    }
}

```

Из рассмотренных примеров можно понять, как задать делегат и как его использовать, но пока не ясно в чем именно преимущество от использования такого инструмента. Основная значимость делегата заключается в возможности абстрагироваться при описании методов в классах от конкретной реализации. Это означает, что методы класса становятся универсальными и могут использоваться как для написания консольного приложения, так и для windows-приложения или web-приложения. Разберем конкретный пример в трех вариантах реализации: консольное приложение без применения делегатов, консольное приложения с делегатом и Windows-приложение с делегатом. И именно наличие такого инструмента как делегат позволит нам с легкостью перенести программный код из консольного приложения в Windows-приложение.

Программа будет получать от пользователя целочисленный параметр count, а в ответ будет выводить (сначала в консоль, а потом и в TextBox) указанное в параметре количество псевдослучайных целых чисел (в диапазоне от 0 до 9 включительно).

Первый вариант реализации наиболее прост, но не имеет возможности для прямого переноса в другой вид приложения:

```
using System;

namespace ООП_delegate_3
{
    public class Utils
    {
        Random r = new Random();
        public void Generate(int count)
        {
            for (int i=0; i<count; i++)
            {
                Console.WriteLine(r.Next(10));
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Utils z = new Utils();
            z.Generate(5);
            Console.ReadLine();
        }
    }
}
```

Итак, у нас есть класс `Utils` с методом `Generate`, который и занимается выводом сгенерированных чисел на экран. Предполагается, что подобного рода методы и классы (имеющие общее для разных программ предназначение) выносятся в отдельные библиотеки, специально созданные для хранения неких универсальных функций (типа, расчет объема шара, вычисления корней квадратного уравнения или, в нашем случае, генерации некоторого количества псевдослучайных чисел). Создание и использование такого рода библиотек классов мы подробно обсудим в последующих главах, а пока, для простоты рассмотрения, класс `Utils` мы оставили вместе с использующим его методом `Main` в одном файле.

Очевидно, что напрямую перенести метод `Generate` в Windows-приложение не удастся из-за отсутствия там класса

Console. Но в Windows-приложении мы смогли бы организовать вывод сгенерированных чисел в любой подходящий объект, например, в TextBox. Для этого нужно обеспечить универсальность метода Generate и абстрагироваться от способа вывода, убрав из метода прямое указание на использование Console.WriteLine. Обратите внимание, что сигнатура метода Console.WriteLine такова void (int), то есть метод на вход получает целочисленное значение, но сам ничего не возвращает. Для того, чтобы получить возможность подставлять в приложении иного типа (Windows, web) другой метод нужно сначала создать делегат с такой же сигнатурой:

```
public delegate void Print(int Num);
```

добавить к аргументам метода Generate переменную типа делегат, для передачи через неё конкретный метод реализации (Console.WriteLine или иной, подходящий для данного типа приложения):

```
public void Generate(int count, Print method)
```

В теле цикла нужно заменить конкретную реализацию Console.WriteLine на абстрактную — method типа делегат:

```
method(r.Next(10));
```

Это мы перечислили только изменения в классе Utils, но предстоит внести также изменения и в тело основной программы. Дополнительно, вне класса Utils, создадим промежуточный метод, реализующий вывод для конкретного типа приложения (в данном случае для консольного):

```
static void Show(int output)
{
    Console.WriteLine(output);
}
```

В теле основной программы создадим переменную со ссылкой на данную реализацию метода и эту переменную наряду с параметром count будем передавать в метод Generate. Ниже приведён полный текст программы со всеми изменениями:

```
using System;
```

```
namespace ООП_delegate_3
```

```
{
    public delegate void Print(int Num);
```

```

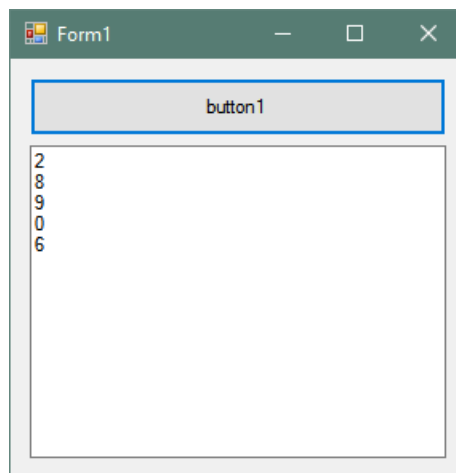
public class Utils
{
    Random r = new Random();
    public void Generate(int count, Print method)
    {
        for (int i=0; i<count; i++)
        {
            method(r.Next(10));
        }
    }
}

class Program
{
    static void Show(int output)
    {
        Console.WriteLine(output);
    }

    static void Main(string[] args)
    {
        Utils z = new Utils();
        Print metod = Show;
        int count = 5;
        z.Generate(count, metod);
        Console.ReadLine();
    }
}

```

В целом изменений немного, однако, теперь данное приложение не составит труда перенести в приложение Windows Forms. Для большего понимания сначала покажу внешний вид приложения:



и программный код с минимальными изменениями:

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

namespace ООП_delegate_4
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        public delegate void Print(int Num);

        public class Utils
        {
            Random r = new Random();
            public void Generate(int count, Print method)
            {
                for (int i = 0; i < count; i++)
                {
                    method(r.Next(10));
                }
            }
        }

        void Show(int output)
        {
            textBox1.Text += Convert.ToString(output) +
                             Environment.NewLine;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Utils z = new Utils();
            Print metod = Show;
            int count = 5;
            z.Generate(count, metod);
        }
    }
}
```

Обратите внимание, что в данной Windows программе изменения претерпел по отношению к консольной только метод Show, а класс Utils остался без изменений. Это будет особенно важно, когда вы станете программировать библио-

теки классов. Напоминаю, что данному вопросу посвящены следующие главы.

Теперь перейдем к ещё одному важному способу использования делегатов, к *событиям*, на которых и основано всё событийное программирование. И, для начала, откажемся от предварительного описания делегата (`public delegate void Print(int Num)`), перейдя к использованию так называемого обобщенного делегата `Action<int>`. Оба вида делегата задают одну и ту же сигнатуру, поэтому не имеет смысла для каждого подобного случая описывать свою собственную сигнатуру и можно воспользоваться способом обобщенных делегатов:

```
using System;

namespace ООП_delegate_3
{
    //public delegate void Print(int Num);

    public class Utils
    {
        Random r = new Random();
        public void Generate(int count, Action<int> method)
        {
            for (int i=0; i<count; i++)
            {
                method(r.Next(10));
            }
        }
    }

    class Program
    {
        static void Show(int output)
        {
            Console.WriteLine(output);
        }

        static void Main(string[] args)
        {
            Utils z = new Utils();
            Action<int> metod = Show;
            int count = 5;
            z.Generate(count, metod);
            Console.ReadLine();
        }
    }
}
```



```
}
```

Продолжим дальше трансформировать наше консольное приложение. Допустим, что нам нужно вызывать метод Generate, но нет необходимости выводить на печать все случайные числа. Ведь их может быть несколько тысяч, мы вполне можем просто заполнить ими массив без отображения его на экране:

```
using System;

namespace ООП_delegate_3
{
    public class Utils
    {
        Random r = new Random();
        public Action<int> Printing { get; set; }
        public void Generate(int count)
        {
            int[] res = new int[count];
            for (int i=0; i<count; i++)
            {
                res[i] = r.Next(10);
                if (Printing!=null)
                    Printing(res[i]);
            }
        }
    }

    class Program
    {
        static void Show(int output)
        {
            Console.WriteLine(output);
        }

        static void Main(string[] args)
        {
            Utils z = new Utils();
            Action<int> metod = Show;
            z.Printing = Show;
            int count = 5;
            z.Generate(count);
            Console.ReadLine();
        }
    }
}
```

В данной программе можно отказаться от вывода последовательности случайных чисел, просто закомментировав строчку:

```
z.Printing = Show;
```

Таким образом, свойству `Printing` просто не будет передана ссылка на реализующий метод и в теле класса после анализа (`if (Printing!=null)`) будет принято решение о необходимости вывода на экран. Между тем сами случайные числа будут генерироваться в любом случае, заполняя массив.

Остался последний шаг для перехода к событиям. Определимся, что события можно объявлять как без параметров, так и с параметрами:

```
public event EventHandler EventName  
public event EventHandler<EventArgs> EventName
```

В нашем случае параметр нужен – это сгенерированное случайное число. В очередной раз внесем некоторые изменения в программу, добавив свой класс `PrintingEventArgs`, являющийся наследником стандартного класса `EventArgs`:

```
public class PrintingEventArgs : EventArgs  
{  
    public int Num { get; private set; }  
    public PrintingEventArgs(int num)  
    {  
        Num = num;  
    }  
}
```

Новый класс будет отвечать за передачу параметра в событии. Уточню, что в классе подобного рода нужно на каждый передаваемый параметр создавать свойство, в данном случае это:

```
public int Num { get; private set; }
```

К текущему времени уже пора привести полный текст программы, чтобы обсуждать финальную реализацию:

```
using System;
```

```
namespace ООП_delegate_3
```

```
{  
    public class PrintingEventArgs : EventArgs  
    {  
        public int Num { get; private set; }  
    }  
}
```

```

        public PrintingEventArgs(int num)
        {
            Num = num;
        }
    }

    public class Utils
    {
        Random r = new Random();

        public event EventHandler<PrintingEventArgs> Printing;

        public void Generate(int count)
        {
            int[] res = new int[count];
            for (int i=0; i<count; i++)
            {
                res[i] = r.Next(10);
                if (Printing!=null)
                    Printing(this,
                        new PrintingEventArgs(res[i]));
            }
        }
    }

    class Program
    {
        static void num_Printing(object sender,
                                PrintingEventArgs e)
        {
            Console.WriteLine(e.Num);
        }

        static void Main(string[] args)
        {
            Utils z = new Utils();
            z.Printing += num_Printing;
            int count = 5;
            z.Generate(count);
            Console.ReadLine();
        }
    }
}

```

Вернемся теперь к обсуждению способа описания свойства Num в классе `PrintingEventArgs`. Обратите внимание на то, что в аксессоре данного свойства для безопасности сеттер сделан приватным. Таким образом, параметр передается

только при вызове данного события прямо в конструкторе класса, что можно увидеть в этой строке кода:

```
Printing(this, new PrintingEventArgs(res[i]));
```

А вот получаем значение параметра мы через соответствующее свойство:

```
Console.WriteLine(e.Num).
```

Если была оформлена подписка объекта *z* на событие `num_Printing` в строке кода:

```
z.Printing += num_Printing,
```

то, по аналогии с предыдущей программой, будет происходить печать случайных чисел.

В событии `num_Printing` два аргумента:

- в переменную `sender` передается объект из переменной `this`;

- в переменную `e` передается объект, сформированный конструктором нашего нового класса `PrintingEventArgs`, но так как конструктору передается значение `res[i]`, то именно поэтому `e.Num` позволяет нам в дальнейшем вывести значение на экран.

Если же в нашей программе нет необходимости передавать ссылку на объект, а достаточно только вызывать соответствующее событие, то можно несколько упростить программу, оставив событие, но вернувшись к использованию обобщенных делегатов:

```
using System;

namespace ООП_delegate_3
{
    public class PrintingEventArgs : EventArgs
    {
        public int Num { get; private set; }
        public PrintingEventArgs(int num)
        {
            Num = num;
        }
    }

    public class Utils
    {
        Random r = new Random();

        public event Action<int> Printing;
    }
}
```

```

        public void Generate(int count)
        {
            int[] res = new int[count];
            for (int i=0; i<count; i++)
            {
                res[i] = r.Next(10);
                if (Printing!=null)
                    Printing(res[i]);
            }
        }
    }

    class Program
    {
        static void num_Printing(int Num)
        {
            Console.WriteLine(Num);
        }

        static void Main(string[] args)
        {
            Utils z = new Utils();
            z.Printing += num_Printing;
            int count = 5;
            z.Generate(count);
            Console.ReadLine();
        }
    }
}

```

Здесь я уже не буду подробно описывать все классы и методы, вам достаточно будет внимательно посмотреть внесенные мною изменения, касающиеся использования обобщенных делегатов. Обратите внимание также на упрощение в вызове события: `num_Printing(int Num)`.

Завершая описание, уточню, что вы можете применять обе формы (полную и сокращенную) описания событий, но при использовании сокращенной формы пропадает возможность получения ссылки на класс издатель и ваша библиотека классов теряет свою универсальность. В тексте данной главы мы уже несколько раз обращались к понятию «библиотека классов» и, так как на текущий момент уже исследованы все базовые вопросы объектно-ориентированного программирования, то можно переходить к расширенным способам работы.

Глава 10. Библиотеки классов

Библиотека классов – это отдельный от основной программы файл, в который выносятся некоторые классы с их свойствами и методами. Как правило, в библиотеку «складывают» те части кода, которые удобно использовать, обращаясь к ним многократно из основной программы или даже из других программ. Библиотека классов предварительно компилируется, но не в исполняемый (executable) файл с расширением *.exe, а в динамически подключаемую библиотеку с расширением *.dll (от англ. Dynamic Link Library – дословно «библиотека динамической компоновки»). Библиотека сама по себе не может запускаться на исполнение, к ней могут обращаться различные программы и использовать подпрограммы, расположенные в классах библиотеки. В операционной системе Microsoft Windows такие библиотеки очень широко распространены и допускают своё использование различными программными приложениями одновременно.

Перечислим некоторые преимущества от применения технологии динамически подключаемых библиотек.

Эффективное использование ресурсов оперативной памяти и снижение объема расходуемого дискового пространства, за счет использования одного экземпляра библиотечного модуля для различных приложений.

Повышение эффективности сопровождения и обновления программных продуктов за счёт их модульности.

Устранение «багов» (ошибок кода, выявленных уже после окончания проектирования) и обновление или наращивание приложений путем замены только динамически подключаемых библиотек с одной версии на другую.

Использование динамических библиотек разнотипными приложениями от одного или даже разных производителей – например, Microsoft Office и Microsoft Visual Studio.

Последовательность работ по формированию библиотеки, как и в других языках программирования, может быть следующей: сначала мы создаем и апробируем часть кода в теле основной программы, а уже потом выносим в отдельную библиотеку. В данном пособии рассмотрим создание и использование библиотеки классов на языке C#.

Предваряя создание библиотеки классов, следует вспомнить о понятии пространства имен. Если раньше я опускал определение пространства имен, так как в этом не было насущной необходимости, то при написании библиотеки классов без этого трудно обойтись. Дело в том, что создавая библиотеку классов нельзя быть уверенным, что имена ваших классов или методов не будут совпадать с именами из других библиотек (стандартных или сторонних производителей), в том числе и ваших собственных в предыдущих и, особенно, в последующих разработках. Именно по этой причине обязательно следует определять пространство имен хотя бы для динамической библиотеки. При выполнении данного требования во всех последующих работах, в случае совпадения идентификаторов, достаточно будет просто уточнить пространство имен, в котором находится интересующая вас функция или даже класс.

Исследуйте следующую программу:

```
using System;

namespace dll_00_main
{
    class Console
    {
        public static void Write(int x)
        {
```

```

        if (x % 2 == 0)
            System.Console.Write("четное");
        else
            System.Console.Write("нечетное");
    }
}
class Program
{
    static void Main(string[] args)
    {
        int e = 12;
        System.Console.Write(e + " - ");
        dll_00_main.Console.Write(e);
        System.Console.ReadKey();
    }
}
}

```

Обратите внимание, что в данной программе я создал класс Console и метод Write, идентификаторы которых в точности совпадают со стандартными наименованиями от C#. Однако я смог воспользоваться своими классом и методом наряду со стандартными. Для разрешения перекрестного использования имен достаточно было просто уточнить в программе, где именно я обращаюсь к своим разработкам, а в каких местах – к стандартным:

```

System.Console.Write(e + " - ");
dll_00_main.Console.Write(e);

```

Из примера видно, что сначала нужно указывать пространство имен и далее через точку выбирать класс и метод. В дальнейшем изложении я постараюсь избегать совпадения имен, чтобы не перегружать программы излишней сложностью.

Перейдем к написанию библиотеки классов и начнем с максимально простого примера – в библиотеку вынесем класс, решающий задачу определения четности числа. Метод класса сделаем статическим, что даст возможность не создавать объект данного класса, а напрямую обращаться к методу класса. Это распространенный подход, например, так

оформлен вывод на экран методом WriteLine, принадлежащим классу Console: `Console.WriteLine("четное");`.

```
using System;

namespace dll_00_main
{
    class Even
    {
        public static bool b(int x)
        {
            return x % 2 == 0;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            int e = 12;
            Console.Write(e + " - ");
            if (Even.b(e))
                Console.WriteLine("четное");
            else
                Console.WriteLine("нечетное");
            Console.ReadKey();
        }
    }
}
```

В данной программе создан класс с одним единственным публичным методом `b(int x)`, возвращающим логическое значение и являющимся статическим. В теле функции `Main` я обращаюсь к данному методу `Even.b(e)` для принятия решения о выводе соответствующей строки на экран.

Обратите внимание, что я оставил инструкцию для определения пространства имен `namespace dll_00_main`. Но большее значение это имеет не для самой программы, а для библиотеки классов, к созданию которой мы сейчас и приступим.

Закройте с сохранением консольное приложение, которое мы исследовали (мы к нему ещё вернемся), и через меню

(Файл / Создать / Проект / Библиотека классов) создайте пустой шаблон под библиотеку классов:

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

namespace DLL_00
{
    public class Class1
    {
    }
}
```

Имя, указанное при создании проекта, будет определять пространство имен. Уточню, что не используемые в данном проекте директивы using можно убрать, а класса переименовать:

```
using System;

namespace DLL_00
{
    public class Even
    {
    }
}
```

Осталось дело за малым, наполнить класс методами или даже добавить классы в пространство имен:

```
using System;

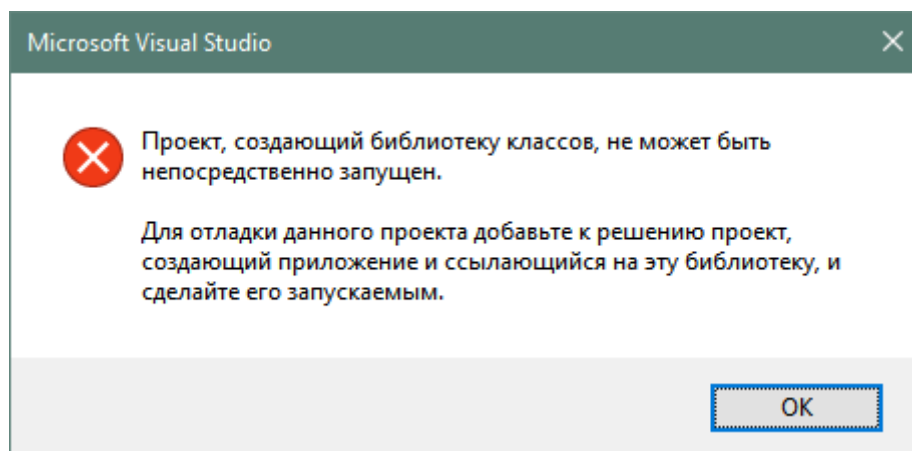
namespace DLL_00
{
    public class Even
    {
        public static bool b(int x)
    }
}
```

```

    {
        return x % 2 == 0;
    }
}

```

Попробуйте запустить данный код на исполнение, нажав клавишу F5, компиляция проекта произойдет, но так как нет функции Main, то компилятор выдаст на экран следующее сообщение:



Очевидно, что методы библиотеки можно использовать только в рамках обращений к ним из другой программы. Сама же откомпилированная библиотека находится в папке текущего проекта библиотеке: `..\DLL_00\DLL_00\bin\Debug`. Найдите файл с расширением `*.dll`, в моем случае это — `DLL_00.dll`. Этот файл можно оставить на месте, но я для удобства скопирую его в папку с основной программой (далее при создании и использовании библиотек классов поступайте по своему усмотрению). Осталось только правильно подключить библиотеку к основной программе. Откройте предыдущую программу, которая использовала обращение к методу `Even.b(e)` для принятия решения о выводе соответствующей строки на экран, и удалите класс `Even` из программы:

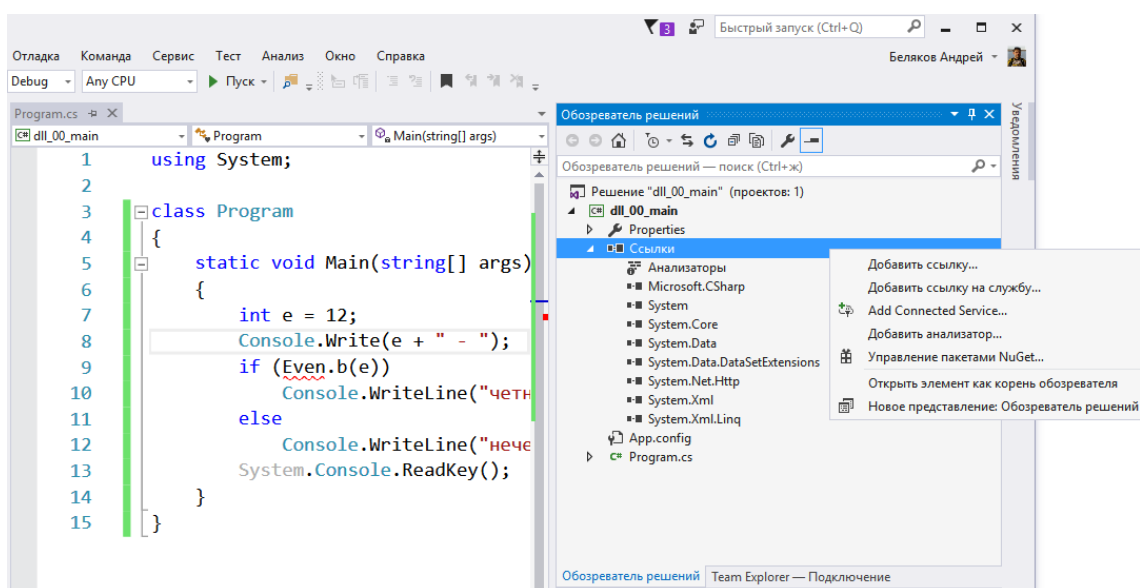
```

using System;

class Program
{
    static void Main(string[] args)
    {
        int e = 12;
        Console.Write(e + " - ");
        if (Even.b(e))
            Console.WriteLine("четное");
        else
            Console.WriteLine("нечетное");
        System.Console.ReadKey();
    }
}

```

Как видите, я даже удалил инструкцию об определении пространства имен за ненадобностью. Однако компилятор предупреждает красным подчеркиванием, что класс Even не определен. Чтобы использовать классы и методы библиотеки DLL_00.dll следует в обозревателе решений добавить ссылку на нашу библиотеку (нажмите правой клавишей мыши на опцию «Ссылки»), через опцию «Обзор» найдите библиотеку и кликните «ОК».



После добавления ссылки на библиотеку возможны два варианта использования классов и методов из неё: либо вы заранее в заголовке через инструкцию `using` указываете пространство имен (что удобно когда нет пересечения имен с другими библиотеками):

```
using System;
using DLL_00;

class Program
{
    static void Main(string[] args)
    {
        int e = 12;
        Console.Write(e + " - ");
        if (Even.b(e))
            Console.WriteLine("четное");
        else
            Console.WriteLine("нечетное");
        Console.ReadKey();
    }
}
```

либо пишете пространство имен непосредственно перед используемым методом:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int e = 12;
        Console.Write(e + " - ");
        if (DLL_00.Even.b(e))
            Console.WriteLine("четное");
        else
            Console.WriteLine("нечетное");
        Console.ReadKey();
    }
}
```

На этом описание технологии создания и подключения динамической библиотеки можно считать завершенным. Приведу краткий список необходимых действий, состоящий всего из пяти шагов:

- создать файл – пустой шаблон библиотеки;
- наполнить его классами и методами, оставив пространство имен;
- откомпилировать его в файл с расширением *.dll;
- положить файл библиотеки в папку с основной программой;
- в основной программе установить ссылку на библиотеку и приступить к использованию методов библиотеки.

Однако при реальном использовании проектов, нужно понимать, что динамические библиотеки они потому «динамические», что их методы подключаются только на этапе исполнения программы (не компилируются в общий exe-файл). Попробуйте забрать исполняемый файл программы (тот, что с расширением *.exe), перенести его на другой компьютер или хотя бы в другую папку и там запустить. Вы получите сообщение об ошибке: «не удалось загрузить файл или сборку». Для исправления этой ситуации перенесите файл с библиотекой `DLL_00.dll` в ту же папку, в которой теперь располагается программа (в моём случае это файлы `dll_00_main.exe` и `DLL_00.dll`). После данного действия повторный запуск основной программы пройдет без осложнений.

Вернемся к развитию вариантов использования созданной библиотеки. Мы создали класс с одним статическим методом, который позволяет обращаться к себе напрямую, минуя объекты. Давайте доработаем библиотеку, отказавшись от статических методов:

```

using System;

namespace DLL_00
{
    public class Even
    {
        public int x { get; set; }
        public Even()
        {
            this.x = 0;
        }
        public Even(int x)
        {
            this.x = x;
        }
        public bool b()
        {
            return this.x % 2 == 0;
        }
    }
}

```

Итак в обновленной библиотеке остался класс Even, но он дополнен свойством x, двумя конструкторами (по умолчанию Even() и с параметром Even(int x)) и методом b(), возвращающим логическое значение. После перекомпиляции посмотрим изменения в способе использования данной библиотеки:

```

using System;
using DLL_00;

class Program
{
    static void Main(string[] args)
    {
        int e = 15;
        Even d = new Even(e);
        Console.Write(d.x + " - ");
        if (d.b())
            Console.WriteLine("четное");
        else
            Console.WriteLine("нечетное");
        Console.ReadKey();
    }
}

```

Теперь мы можем создать объект (или даже множество разных объектов) типа Even

```
Even d = new Even(e);
```

и воспользоваться в нужный момент возможностями соответствующего класса:

```
if (d.b())  
    Console.WriteLine("четное");  
else  
    Console.WriteLine("нечетное");
```

В данном случае метод b() объекта d возвращает true, если поле x четное и false в ином случае.

Отмечу, что для удобства разработки и отладки имеет смысл открыть основной проект и библиотеку в двух различных экземплярах Visual Studio (запустить программу Visual Studio два раза) и работать в них параллельно.

Теперь уже можно переходить к изучению подходов к разработке более сложных библиотек с обработкой строк, массивов или списков.

Глава 11. Практика разработки библиотек классов

Библиотека классов не обязана содержать только какие-то конкретные методы для решения одной задачи, это могут быть различные удобные функции широкого назначения или, например, элементы оформления интерфейса.

Пример 1.

Рассмотрим возможный вариант оформления функции завершения работы программы. Добавьте в пространство имен нашей библиотеки DLL_00 открытый класс Utils, снабдив его статической функцией Pause:

```
public class Utils
```



```

{
    public static void Pause()
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

```

Тогда в основной программе вместо избитого подхода:

```
Console.ReadKey();
```

можно будет использовать интереснее оформленный:

```
Utils.Pause();
```

Более того, ввиду возможности перегрузки функций, мы можем создать перегружаемую версию Pause() с параметром, в котором будем передавать клавишу, определяющую закрытие программы:

```

public static void Pause(string e)
{
    ConsoleKey t;
    string[] arr = { "Escape" , "Space" , "Enter" };
    string st = "";
    switch (e)
    {
        case "Escape":
            st = arr[0];
            t = ConsoleKey.Escape;
            break;
        case "Space":
            st = arr[1];
            t = ConsoleKey.Spacebar;
            break;
        default:
            st = arr[2];
            t = ConsoleKey.Enter;
            break;
    }
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("Press " + st + " key...");
    do
    { } while (Console.ReadKey(true).Key != t);
}

```

В приведенном примере упомянуты только три возможные клавиши ("Escape" , "Space" , "Enter"), но ничто вам не мешает при желании расширить этот список. Параметр true в инструкции `Console.ReadKey(true).Key` определяет отсутствие вывода нажатой клавиши в консоль. В последних строках метода `Pause(string e)` организован цикл, ожидания нажатия установленной клавиши, после чего основная программа будет закрыта. Порядок обращения данному методу из основной программы такой:

```
Utils.Pause("Escape");
```

Очевидно, что если параметр метода будет набран с ошибками или будет передана пустая строка, то будет выполнена ветвь `default` оператора многоальтернативного выбора, и программа будет реагировать только на нажатие клавиши Enter. Если же параметра не будет вовсе, то сработает перегруженный метод `Pause()` без параметра и будет реализован функционал выхода из программы по нажатию на любую клавишу.

Пример 2.

Следующий пример показывает, как можно оформить вывод вещественного числа, задав необходимое количество отображаемых после запятой знаков. Добавьте в открытый класс `Utils` статическую функцию `ToFormat`:

```
public static string ToFormat(int count)
{
    string format = "{0}";
    return format.Insert(
        format.IndexOf('0') + 1, ":F" + count.ToString());
}
```

Формат обращения к данной функции показан в следующем примере:

```
using System;
using DLL_00;

class Program
{
    static void Main(string[] args)
    {
        Random r = new Random();
        double n = r.NextDouble();
        Console.WriteLine(n);
        int count = 4;
        Console.WriteLine(Utils.ToFormat(count), n);
        Utils.Pause();
    }
}
```

В данной программе генерируется псевдослучайное вещественное число и выводится в консоль с точностью, доступной типу `double`, а затем – с точностью заданной пользователем через переменную `count` и сформированной функцией `ToFormat`.

Если вы регулярно разрабатываете проекты различного рода, то легко сможете выделить для себя подмножество тех функций, которые из раза в раз приходится реализовывать. Полагаю, что рассмотренная технология создания динамических библиотек, с учетом последних двух примеров, подстегнет ваш энтузиазм к самостоятельному развитию и расширению функционала нашей библиотеки `DLL_00.dll`.

Пример 3.

Одним из востребованных направлений формирования библиотек классов является разработка функционала, связанного с парсингом строк. Например, можно добавить функцию, которая исправляет набор в неправильной раскладке. Если вы набирали русскую фразу «раскладка клавиа-

туры», но забыли переключиться с английской раскладки на русскую, то в результате получите такой текст «hfcrkflrf rkfdbfnehs». Такая ситуация происходит довольно часто с теми, кто использует оба языка. Некоторые редакторы предлагают функцию исправления «неправильной» раскладки. Давайте и мы попробуем создать свою реализацию данной функции в нашей библиотеке.

Логика работы данной функции достаточно проста: нужно последовательно пройти все символы в строке, введенной в неправильной раскладке, и заменить их на символы из противоположной раскладки. Добавьте в класс Utils два открытых метода для доступа из основной программы и закрытые поле и функцию:

```
static string[] st =
{ @"qwertyuiop[]asdfghjkl;'zxcvbnm,.",
  @"йцукенгшщзхъфывапролджэячсмитьбю" };

static string _conv(string tmp, int a, int b)
{
    int pos = 0; string result = "";
    for (int i = 0; i < tmp.Length; i++)
    {
        pos = st[a].IndexOf(tmp[i]);
        result += st[b].Substring(pos, 1);
    }
    return result;
}

public static string conv(string tmp)
{
    int a = 0, b = 1;
    return _conv(tmp, a, b);
}

public static string conv(string tmp, byte n)
{
    int a = 0, b = 1;
    if (n > 0)
        { a = 1; b = 0; }
    return _conv(tmp, a, b);
}
```

Поле `st` хранит массив из двух строк, задающих соответствие между символами английской и русской раскладок клавиатуры. Закрытый метод `_st` содержит цикл, который последовательно перебирает все символы введенной пользователем строки, находит их позиции в соответствующей раскладке и заменяет их на символы из другой раскладки:

```
for (int i = 0; i < tmp.Length; i++)
{
    pos = st[a].IndexOf(tmp[i]);
    result += st[b].Substring(pos, 1);
}
```

Для пользователя предоставлены две перегрузки метода `conv`, отличающиеся количеством атрибутов. Если один атрибут, то можно передавать только «неправильную» строку и метод будет её конвертировать и английской раскладки в русскую. Если же нужно установить направление перекодирования, то нужно использовать перегруженный метод `conv(string tmp, byte n)`, где в качестве второго аргумента можно ввести целое число (0 – из английской в русскую, 1 – наоборот):

```
int a = 0, b = 1;
if (n > 0)
    { a = 1; b = 0; }
return _conv(tmp, a, b);
```

Порядок обращения из основной программы такой:

```
using System;
using DLL_00;

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

```

        string s = Console.ReadLine();
        Console.WriteLine(Utils.conv(s,1));
        //Console.WriteLine(Utils.conv(s));
        Utils.Pause("");
    }
}

```

Пример 4.

Продолжим работать со строками и рассмотрим один поучительный пример анализа структуры строки. Для интереса будем имитировать стиль разговора небезызвестного мастера Йоды из саги «Звёздный войны», постоянно инвертирующего последовательность слов в предложении. Например, нормально построенную фразу «ты должен понять суть явлений», Йода говорит так: «явлений суть понять должен ты». Есть, конечно, несколько уточнений к данному алгоритму, но мы не будем углубляться дальше обычной инверсии слов во фразе.

Рассмотрим первый вариант реализации функции `yoda_1`

```

public static string yoda_1(string text)
{
    string[] t = text.Split(' ');
    Array.Reverse(t);
    return String.Join(" ", t);
}

```

В данной функции используется метод `Reverse`, для его работы необходимо подключить в нашей библиотеке `Utils` пространство имен:

```
using System.Linq;
```

Логика работы разработанной функции заключается в том, чтобы сначала разбить строку на массив слов (`string[] t = text.Split(' ');`), затем развернуть его (`Array.Reverse(t);`) и сформировать из него новую строку с обратным следованием

слов (`String.Join(" ", t);`). Если смысл работы этой функции ясен, то не будет затруднений в понимании её сокращенного варианта:

```
public static string yoda_2(string text)
{
    return String.Join(" ", text.Split(' ').Reverse());
}
```

Пример 5.

Продолжим исследовать методы обработки строк и добавим в нашу библиотеку метод `selsort`, который будет фильтровать заданный массив строк (например, фамилии студентов) по заданной первой букве и возвращать полученный список отсортированным по алфавиту. Так как мы будем использовать класс список (`List`), то следует в заголовочной части нашей библиотеки подключить пространство имен:

```
using System.Collections.Generic;
```

Метод `selsort` состоит из трех частей. Сначала в цикле из исходного массива отбираются только те слова, которые соответствуют заданному условию, затем сформированный список сортируется, на завершающем этапе из списка формируется массив строк:

```
public static string[] selsort(string[] arr, string begin)
{
    var selectedWords = new List<string>();
    foreach (string s in arr)
    {
        if (s.ToUpper().StartsWith(begin))
            selectedWords.Add(s);
    }
    selectedWords.Sort();
    string[] temp = new string[selectedWords.Count];
    for (int i=0; i<selectedWords.Count; i++)
        temp[i] = selectedWords[i];
    return temp;
}
```

Порядок обращения из основной программы такой:

```
using System;
using DLL_00;

class Program
{
    static void Main(string[] args)
    {
        string[] names = { "Бунин", "Воронин", "Ульянов",
                           "Мишин", "Ардов", "Беляков", "Букин", "Иванов" };
        string[] otbor;
        string begin = "Б";
        otbor = Utils.selSort(names, begin);
        foreach (string s in otbor)
            Console.WriteLine(s);

        Utils.Pause("");
    }
}
```

В результате работы данной программы в консоль будут выведены строки:

```
Беляков
Букин
Бунин
Press Enter key...
```

Предположим необходимо решить следующую практическую задачу. Есть текстовый файл, в котором фиксируются даты подачи документов студентами в приемную комиссию вуза. Необходимо выбрать студентов, чьи фамилии начинаются на заданную букву, и сформировать из них в текущей папке файл с отсортированным списком фамилий.

Формат входного файла: записи идут построчно, в каждой строке сначала дата, затем через символ табуляции фамилия, например:

```
10.07.17   Бунин
11.07.17   Букин
```



```
11.07.17    Петров
12.07.17    Беляков
15.07.17    Иванов
16.07.17    Афонин
```

Так как библиотечная функция `selsort(string[] arr, string begin)` уже готова, то остается только корректно подготовить входные данные, которые находятся в файле построчно совместно с другими данными. Возможный вариант организации может быть таким:

- читаем файл построчно,
- каждую строку разбиваем на массив строк по разделителю (символ табуляции),
- выбираем только фамилии (это будет второй элемент массива строк с индексом «1»),
- формируем из фамилий отдельный массив,
- отправляем массив фамилий на обработку в функцию `selsort`, совместно с критерием отбора (первый символ в фамилии, который определяет фильтрацию),
- полученный из функции массив выводим в выходной файл (имя файла формируется на основе имени входного файла с добавлением символа, с которого начинаются отбираемые фамилии, например, если входной файл «**Группа ИСб-11.txt**» и фильтр «Б», то выходной – «**Группа ИСб-11Б.txt**»).

Обратите внимание, что для реализации ряда методов обработки файлов и работы со строками необходимо добавить пространства имен (`System.IO` и `System.Text`):

```
using System;
using System.IO;
using System.Text;
using DLL_00;

class Program
```

```

{
    static void Main(string[] args)
    {
        char char_del = (char)9;
        string FileName = "Группа ИС6-11.txt";
        string[] arrStr =
            File.ReadAllLines(FileName, Encoding.UTF8);
        string[] sline;
        int stud_cnt = arrStr.Length;
        string[] names = new string[stud_cnt];
        for (int i = 0; i < stud_cnt; i++)
        {
            sline = arrStr[i].Split(char_del);
            names[i] = sline[1];
        }

        string begin = "Б";
        string[] otbor = Utils.selSort(names, begin);
        string[] srt = new string[otbor.Length];
        FileName = FileName.Insert(FileName.IndexOf('.'), begin);
        File.WriteAllLines(FileName, otbor);

        Utils.Pause("");
    }
}

```

Таким образом, единожды разработанную библиотечную функцию можно использовать многократно в разных программах и для решения различных практических задач. В этом смысле может оказаться весьма полезной функция подсчета количества (частоты) каждой присутствующей буквы в тексте.

Добавим в наш класс статический метод, в качестве аргумента принимающий строку, а возвращающий целочисленный массив частот символов:

```

public static int[] countChar(string text)
{
    text = text.ToLower();
    // массив для подсчета частот
    int[] c = new int[(int)char.MaxValue];
    // перебор всех символов
    foreach (char t in text)
        c[(int)t]++;
}

```

```
        return c;
    }
}
```

Порядок обращения к библиотечной функции достаточно прост, формируем строку и передаем её в функцию countChar:

```
int[] c = Utils.countChar(text);
```

Некоторые сложности могут возникнуть при формировании вывода результата, ниже предложен один из возможных вариантов:

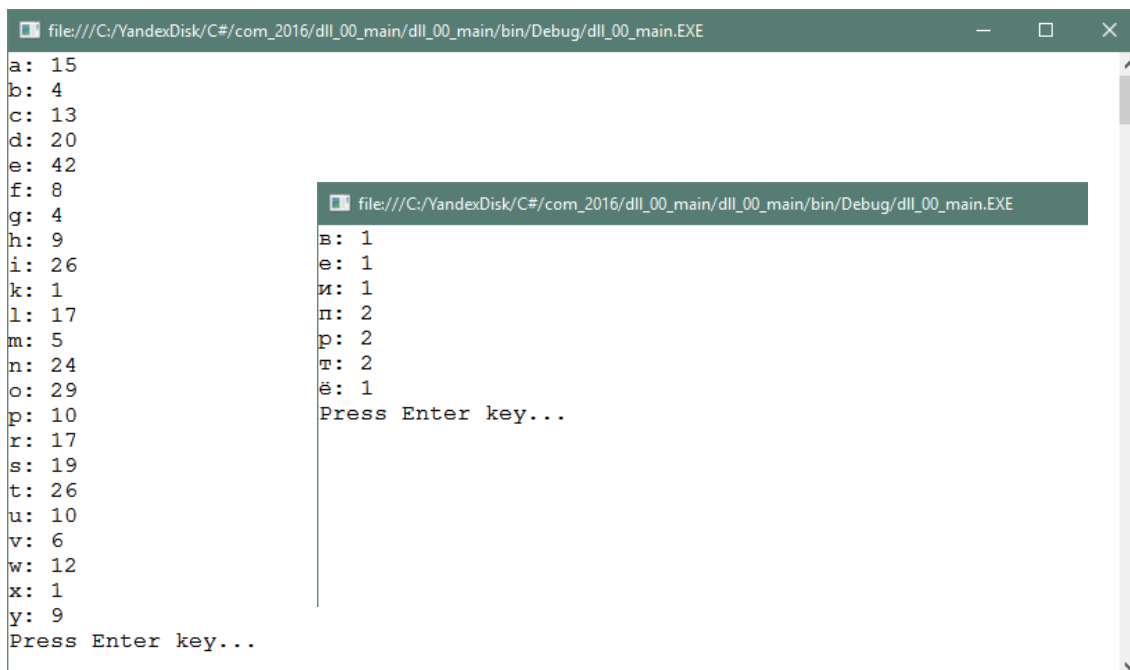
```
using System;
using System.IO;
using System.Text;
using DLL_00;

class Program
{
    static void Main(string[] args)
    {
        //string text = "Event description: Further develop your
        knowledge of modern Windows deployments by exploring the different
        technical options available to your customers and discovering the
        capabilities of Windows-as-a-Service (WaaS). Building on the
        deployment features first introduced with Windows 10 at release, the
        deep dive technical content will help you confidently deploy Windows
        to your customers.";
        string text = "Привет, Пётр!";
        int[] c = Utils.countChar(text);
        for (int i = 0; i < (int)char.MaxValue; i++)
            if (c[i] > 0 && char.IsLetter((char)i))
                // вариация для вывода букв и цифр
                //if (c[i] > 0 && char.IsLetterOrDigit((char)i))
                    Console.WriteLine("{0}: {1}", (char)i, c[i]);

        Utils.Pause("");
    }
}
```

Обратите внимание, что в зависимости от поставленной задачи можно преобразовать код программы так, чтобы на экран выводились частоты только цифр, только букв (можно отделить русские от английских) или сочетание и букв и цифр.

В тексте программы я оставил стоку с английским текстом и с русским. Ниже можно посмотреть, как формируется вывод данной программой для обеих строк:



```
file:///C:/YandexDisk/C#/com_2016/dll_00_main/dll_00_main/bin/Debug/dll_00_main.EXE
a: 15
b: 4
c: 13
d: 20
e: 42
f: 8
g: 4
h: 9
i: 26
k: 1
l: 17
m: 5
n: 24
o: 29
p: 10
r: 17
s: 19
t: 26
u: 10
v: 6
w: 12
x: 1
y: 9
Press Enter key...

file:///C:/YandexDisk/C#/com_2016/dll_00_main/dll_00_main/bin/Debug/dll_00_main.EXE
в: 1
е: 1
и: 1
п: 2
р: 2
т: 2
ё: 1
Press Enter key...
```

Все вопросы проектирования библиотек классов в рамках одного пособия рассмотреть затруднительно, да и не имеет большого смысла, так как в настоящее время имеется значительно количество динамических библиотек, разработанных сторонними производителями. При желании можно найти в сети Интернет библиотеку, необходимую для решения частной задачи. Давайте оценим, насколько может быть сложным использование «чужой» библиотеки.

Глава 12. Библиотеки классов сторонних производителей

Разработка собственной библиотеки классов увлекательное и практически значимое занятие, однако, в некоторых случаях не стоит «изобретать велосипед» и имеет смысл в сети Интернет поискать готовое решение от сторонних производителей. Под платформу .NET в настоящее время можно найти библиотеки классов в буквальном смысле на все случаи в жизни. Давайте попробуем разработать приложение по работе с QR-кодами.

Под QR-кодом (англ. quick response – быстрый отклик) понимают рисунок (квадратное изображение, по сути, двумерный штрихкод) с зашифрованной в нём информацией. Основное достоинство QR-кода заключается в простоте распознавания его сканирующим оборудованием, что дает возможность использования в торговле, производстве, логистике и даже в туризме и музейном деле.

Хотя обозначение «QR code» является зарегистрированным товарным знаком «DENSO Corporation» (разработали в середине девяностых годов), использование кодов не облагается никакими лицензионными отчислениями, а сами они описаны и опубликованы в качестве стандартов ISO. Существует несколько кодировок QR-кодов и несколько уровней коррекции ошибок распознавания. Алгоритмы генерации и чтения QR-кода нетривиальны и нет особого смысла самостоятельно писать программу чтения и генерации кода при наличии свободно распространяемых библиотек.

Для обучения воспользуемся одной из простых, но бесплатных библиотек – `MessagingToolkit.QRCode.dll`, которая позволит обработать только 66 символов в кириллице или 122 в латинице. Если возникнет необходимость работать с

большим объемом текста, то следует искать бесплатные или платные библиотеки, поддерживающие большой объем данных. В зависимости от используемых способов кодировки и уровней коррекции, заложенных в библиотеке, мы можем получить сильно различающийся функционал, но в лучшем случае максимальное количество символов, которые помещаются в один QR-код, составляет 7089 цифр или 4296 цифр и букв в латинице.

Для начала следует скачать библиотеку, для чего можете самостоятельно поискать её в сети Интернет или воспользоваться следующей ссылкой, сгенерированной Интернет-ресурсом `qrcoder.ru`:



или библиотекой `MessagingToolkit.QRCode.dll`:



Как вы, наверное, заметили, рисунки заметно отличаются, что как раз и объясняется использованием различных способов кодирования и отличающимся уровнем коррекции.

Итак создайте приложение Windows Forms, разместите библиотеку `MessagingToolkit.QRCode.dll` в папке с приложением и поставьте на неё ссылку как мы обсуждали в главе «Библиотеки классов C#», далее добавьте в заголовочной части ссылки на пространства имен библиотеки:

```
using MessagingToolkit.QRCode.Codec;  
using MessagingToolkit.QRCode.Codec.Data;
```

Прежде чем писать код программы обсудим её функционал и примерный дизайн. Наша программа должна уметь кодировать текст в QR-код, декодировать текст из QR-кода, сохранять изображения QR-кода в графический файл и читать графический файл с QR-кодом для последующего декодирования.

Примерный вид дизайна приложения представлен ниже:



Для открытия графического файла и сохранения рисунка с кодом добавьте диалоги открытия (`openFileDialog1`) и сохранения (`saveFileDialog1`). Настройте фильтры диалогов на работу с различными форматами графических файлов:

PNG | *.png | JPEG | *.jpg | GIF | *.gif | BMP | *.bmp.

Предварительные работы закончены, теперь можно установить на форму объект для хранения изображения pictureBox1 и приступить к кодированию обработчиков событий нажатия экранных клавиш. Наиболее просто выглядят коды открытия файла:

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
    pictureBox1.ImageLocation = openFileDialog1.FileName;
```

и сохранения графического файла:

```
if (pictureBox1.Image != null)
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
        pictureBox1.Image.Save(saveFileDialog1.FileName);
```

Обратите внимание, что каждый раз мы проверяем, утвердительно ли пользователь ответил в файловом диалоге, а также при сохранении файла дополнительно проверяем наличие сгенерированного файла с QR-кодом в объекте pictureBox1.

Процесс кодирования текста в QR-код с использованием подключенной библиотеки выглядит не намного сложнее:

```
string txt = textBox1.Text;
QRCodeEncoder coder = new QRCodeEncoder();
//Bitmap qrcode = coder.Encode(textBox1.Text);
Bitmap qrcode = coder.Encode(txt, Encoding.UTF8);
pictureBox1.Image = qrcode as Image;
```

Сначала текст сохраняем в строковую переменную, создаем новый объект для работы с QR-кодом, используем его метод Encode для формирования графического образа кода, далее размещаем рисунок в свойстве Image объекта pictureBox1. Обратите внимание, что есть перегруженный вариант метода

Encode, в котором можно не указывать кодировку текста (я оставил его в комментариях).

Процесс распознавания QR-кода обрамим конструкцией обработки исключительных ситуаций на случай невозможности распознавания кода:

```
try
{
    Bitmap decImg = new Bitmap(pictureBox1.Image);
    QRCodeBitmapImage qrImg = new QRCodeBitmapImage(decImg);
    QRCodeDecoder decoder = new QRCodeDecoder();
    //decoder.decode(qrImg);
    textBox2.Text = decoder.decode(qrImg, Encoding.UTF8);
}
catch (Exception err)
{
    MessageBox.Show(
        "нет распознаваемого образа, ошибка - \n\r" + err);
}
```

На этом процесс создания приложения по работе с QR-кодом можно считать законченным, осталось только найти ему практическое применение. Как вы сами могли убедиться, использование в своей программе библиотек классов от сторонних производителей может быть совсем простым процессом. В большинстве случаев не имеет особого смысла самому писать новую библиотеку классов, а достаточно найти кем-то уже реализованный функционал.

Задание для самостоятельного исполнения по темам

1. Создайте перегружаемую функцию для организации «паузы» экрана консольной программы (вместо `Console.ReadLine(i);`) в двух вариантах исполнения: пользователь может передать в функцию код символа или строку, которые будут обозначать по нажатию какой клавиши снимать пауза, например, код символа – 27 или строка – «Escape».

2. Создайте структуры для хранения координат геометрических фигур: прямоугольника, круга.

3. Создайте класс для перевода целых чисел из десятичной системы счисления в двоичную и обратно.

4. Создайте класс Student с конструктором, инициализирующим поля «факультет» и «направление обучения».

5. Создайте класс Student со скрытым полем Age и открытым методом установки «безопасного» значения этого поля (не может быть меньше 0 и больше 100).

6. Создайте базовый класс Human (с полем Age) и наследуемый Student (с полями «факультет» и «направление обучения»).

7. Создайте базовый класс Human (с полями «Фамилия», «Имя», «Отчество» и виртуальным методом – вывод с инициалами: Фамилия И.О.) и наследуемые классы Student и Teacher (с переопределяемым методом, так чтобы у преподавателя выводилось полностью – Фамилия Имя Отчество, а у студента – Фамилия Имя).

8. Создайте интерфейс «Человек», включающий декларацию метода Name (возвращает строку). Создайте два класса на основе этого интерфейса: «Студент» и «Преподаватель». Конкретизируйте в классе «Студент» метод Name для возвращения имени студента, а в классе «Преподаватель» – фамилии преподавателя.

9. Создайте класс с двумя методами с одинаковой сигнатурой – через аргументы передаётся массив с Фамилией Именем Отчеством сотрудника, а возвращается строка – в первом методе «Фамилия И.О.», во втором – «Фамилия Имя». Создайте делегат для ссылки на эти методы и продемонстрируйте их работоспособность.

10. Разработайте динамическую библиотеку (файл с расширением *.dll) с двумя методами: перевод целых чисел из десятичной системы счисления в двоичную и обратно. Создайте программу для проверки работоспособности библиотеки.

11. Внесите изменения в динамическую библиотеку из задания 10 так, чтобы методы были статическими. Покажите особенности использования статических методов.

12. Доработайте программу главы 12 так, чтобы осуществлялась потоковая обработка файлов, то есть пользователь загружает в программу текстовый файл и указывает папку сохранения QR-кодов, а программа построчно (на каждую строку свой QR-код) из текстового файла сохраняет информацию в виде QR-кодов в указанную папку.

Заключение

Объектно-ориентированное программирование, как процесс, приближается к моделированию реального мира, так как предоставляет программисту такие инструменты как классы и объекты. Принципы объектно-ориентированного программирования, такие как инкапсуляция, наследование и полиморфизм, а также такие инструменты как структуры, интерфейсы и делегаты позволяют логично и последовательно описать необходимые для решения частной задачи сущности.

При описании классов необходимо стремиться к максимальному сокращению области видимости каждого поля, то есть к реализации принципа инкапсуляции. Для систематизации сущностей необходимо предварительно построить иерархию классов, распределить поля и методы между базовыми и производными классами. Для повышения гибкости кода в C# предусмотрен механизм динамического полиморфизма, позволяющий переопределять методы базового класса в производных, а при отсутствии такового динамически подбирать подходящий в ближайшем классе выше по иерархии. Для повышения универсальности кода можно использовать такие инструменты как интерфейсы и делегаты. В свою очередь, сами делегаты позволяют организовать обработку событий объектов, на чем основано событийное программирование. Таким образом, технология объектно-ориентированного программирования зиждется на нескольких взаимосвязанных принципах и инструментах, которые обязательно следует изучать, агрегируя знания и формируя опыт в реализации конкретных приложений.

Язык программирования C# и платформа .NET постоянно развиваются, вносятся исправления и уточнения, дорабатываются интерфейсы и методы, расширяется и упрощается

синтаксис, что требует от разработчика перманентной включенности в процесс самообразования и накопления опыта объектно-ориентированного проектирования и разработки методов и классов, динамических библиотек и основанных на них приложений.

Для наглядности я приведу только один простой пример по созданию и использованию класса `Point` в версиях `C#5.0` (2012 год) и `C#6.0` (2015 год):

```
using System;
using static System.Math; // для C#6.0
using static System.Console; // для C#6.0
using i = System.Int32; // псевдоним

namespace CA_cs_version6
{
    class Program
    {
        public class Point
        {
            public int X { get; set; }
            public int Y { get; set; }
            public Point(int x, int y) { X = x; Y = y; }
            public double Dist
            {
                get { return Math.Sqrt(X * X + Y * Y); }
            }
            public override string ToString()
            {
                return String.Format("{0}, {1}", X, Y);
            }
        }
        public class Point_6
        {
            public int X { get; } = 3;
            public int Y { get; } = 4;
            public Point_6() { }
            public Point_6(i x, i y) { X = x; Y = y; }
            public double Dist => Sqrt(X * X + Y * Y);
            public override string ToString() => $"({X}, {Y})";
        }
        static void Main(string[] args)
        {
            Point a = new Point(3, 4);
            Console.WriteLine(a.ToString());
        }
    }
}
```

```

        Console.WriteLine(a.Dist);

        Point_6 b = new Point_6();
        //Point_6 b = new Point_6(3, 4); //так тоже можно
        WriteLine(b.ToString()); // C#6.0
        WriteLine(b.Dist); // C#6.0

        ReadKey(); // C#6.0
    }
}

```

Если вы смогли одолеть пособие, то данная программа должна быть вам понятна. В представленном коде вы сможете отыскать, сравнивая два эквивалентных класса, особенности синтаксиса новой версии, на которые я укажу далее. Эти изменения носят косметический характер и направлены на упрощение рутинной работы программиста и сокращение объема кода:

- в аксессорах свойств теперь допускается возможность не писать явно сеттеры, что удобно для формирования так называемых «неизменяемых свойств» (в них есть только геттеры);

- появились инициализаторы свойств, что сделало возможным непосредственно в аксессорах обычным присвоением устанавливать значения полей по умолчанию, которые будут задействованы при вызове конструктора без параметров;

- разрешение доступа к статическим методам классов из библиотек классов через их предварительное импортирование директивой `using` (например, `using static System.Math`) без необходимости последующей квалификации доступа с помощью имени типа (вместо `Math.Sqrt` теперь можно так `Sqrt`);

- включение нового синтаксиса по форматированию строк – интерполяция строк, заметно сокращающая рутинное

описание формата (вместо `String.Format("{0}, {1})", X, Y)` теперь можно так: `$"({X}, {Y})"`);

— ввели в использование свойства-выражения, представляющие собой методы, которые вычисляют только одно выражение и возвращают его (например, `public double Dist => Sqrt(X * X + Y * Y)` или `public override string ToString() => $"({X}, {Y})"`).

Несмотря на то, что переходы между предыдущими версиями языка были заметно более кардинальными, с включением новых ярких возможностей и инструментов (именованные и необязательные аргументы, язык интегрированных запросов LINQ, лямбда-выражения, «ленивые выражения» и другие инструменты), но я остановился на данном примере по причине его непосредственной близости к теме учебного пособия.

В данное пособие лишь в малой части попали вопросы посвященные проектированию приложений, основанных на визуальных формах (Windows Forms). Также не рассматривался и вопрос частной разработки визуальных компонентов как самостоятельных классов и правила их использования в проектируемых информационных системах. В пособии основное внимание было сосредоточено на базовых вопросах объектно-ориентированного программирования имеющих общее значение для всех видов проектов: консольных, web-приложений, Windows Forms или Windows Presentation Foundation приложений и любых других, где имеется возможность использовать классы и объекты.

Библиографический список

1. Павловская Т.А. С#. Программирование на языке высокого уровня : учебник для вузов. – СПб. : Питер, 2014. – 432 с.
2. Лафоре Р. Объектно-ориентированное программирование в С++. Классика Computer Science. – 4-е изд. – СПб. : Питер. 2008. – 928 с.
3. Фленов М.Е. Библия С#. – 2-е изд. – СПб. : БХВ-Петербург, 2013. – 560 с.
4. Язык С# и платформа. NET Framework [Электронный ресурс] / автор курса Александр Ерохин, 2016. – режим доступа https://professorweb.ru/my/csharp/charp_theory/level7/7_6.php, свободный. – Загл. с экрана.
5. Агуров П.В. С#. Разработка компонентов в MS Visual Studio 2005/2008. СПб. : БХВ-Петербург, 2008. – 480 с.