Лабораторная работа №05. Простейшие классы в языке C#

1 Цель и порядок работы

Цель работы – ознакомиться с понятием класса и объекта в языке C#.

Порядок выполнения работы:
- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Введение в объектно-ориентированное программирование

Как мы видим окружающий мир? Ответ зависит от нашего прошлого. Ученый может видеть мир как множество молекулярных структур. Художник видит мир как набор форм и красок. А кто-то может сказать, что мир — это множество вещей. Вероятно, первая мысль, которая пришла вам в голову, когда вы прочитали этот вопрос, была: какое значение имеет то, как кто-то видит мир?

Это имеет большое значение для программиста, которому необходимо написать программу, эмулирующую реальный мир.

Мы видим мир как составляющие его вещи, и, возможно, это правильно. Дом — вещь. Имущество, которое находится в доме, — вещи. То, что вы выкидываете, — вещи, покупаете вы вещи. Вещи, которые мы имеем, например, дом, сделаны из других вещей, таких как окна или двери.

Технически говоря, вещь (stuff) — это объект. То есть дом — объект. Вещи, которые находятся в доме, — объекты. Вещи, которые вы выбрасываете, — объекты, и вещи, которые вы хотите купить, — тоже объекты. Все мы, независимо от нашего прошлого, видим окружающий мир как множество объектов. Объект — это: человек, место, вещь, понятие и, возможно, событие.

Лучший способ изучить объекты состоит в том, чтобы исследовать наиболее важный из них — нас самих. В мире объектно-ориентированного программирования каждый из нас рассматривается как объект. Мы называем этот объект — человек. Человек, так же как дом, машина и любой другой объект реального мира, описывается с помощью двух наборов свойств: 1) атрибутов и 2) поведения. Атрибут — это характеристика объекта. Например, у человека есть имя, фамилия, рост и вес. Имя, фамилия, рост и вес — это атрибуты всех людей. Человека характеризуют и сотни других атрибутов, но мы остановимся на этих четырех. Поведение — это действие, которое объект может осуществить. Человек может: сидеть, стоять, идти или бежать, и это не считая тысяч других вариантов поведения.

Кому: Иван Петров
345247, Тюмень
Ямская, 11, кв. 45

Заказ

Номер заказа	Дата заказа	Заказано через	Дата поставки	Срок
AT345	12.02.2013	UPS	14.02.2013	15 дней

Количество	Номер товара	Наименование	Цена	Сумма
4	42	Полотенце	100.00 руб.	400.00 руб.

3	58	Подушка	250.00 руб.	750.00 руб.
	Стоимость товара без налога			1150.00 руб.
	Налог			0.00 руб.
	Доставка			44.00 руб.
	Итого			1194.00 руб.

Рис. 2.1. Форма заказа — это объект, характеризуемый атрибутами и вариантами поведения

Каждый из объектов автомобиль, самолет, документ, форма заказа, характеризуется своими атрибутами и поведением. Атрибуты и варианты поведения автомобиля и самолета довольно очевидны. Оба они имеют ширину, высоту, вес, колеса и двигатель, а также множество других атрибутов. Автомобиль и самолет могут двигаться в некотором направлении, останавливаться и начинать двигаться в другом направлении, также они могут осуществлять сотни других действий (вариантов поведения). Однако сложнее определить атрибуты и варианты поведения документа или формы заказа (рис. 1.1). Атрибутами объекта – форма заказа являются: имя клиента, адрес клиента, заказанный товар, его количество, а также другая информация, которую можно найти в форме заказа. Варианты поведения формы заказа включают сбор информации, модификацию информации и обработку заказа.

Объекты рассматривают двумя способами: 1) как абстрактные объекты и 2) как реальные объекты. Абстрактный объект — это описание реального объекта. Например, абстрактный человек — это описание человека, которое содержит атрибуты и варианты поведения. Вот четыре атрибута, которые могут быть найдены у абстрактного человека (эти атрибуты только идентифицируют тип характеристики, например, имя или вес, но не определяют само имя или значение):

имя,
фамилия,
рост,
вес.

Абстрактный объект используется как модель для реального объекта. Реальный человек имеет все атрибуты и варианты поведения, определенные в абстрактном объекте, а также подробности, упущенные в абстрактном объекте. Например, абстрактный человек — это модель реального человека. Абстрактный человек определяет, что реальный человек должен иметь имя, фамилию, рост и вес. Реальный человек определяет значения, связанные с этими атрибутами, например, такие:

Иван,
Петров,
180 см,
80 кг.

Программисты создают абстрактный объект, а затем используют его для создания реального объекта. Реальный объект называется экземпляром (instance) абстрактного объекта. Можно сказать, что реальный человек — это экземпляр абстрактного человека.

Почему же целесообразнее смотреть на мир как на множество объектов? Фокусирование на объектах упрощает для нас понимание сложных вещей. Объекты позволяют уделять внимание важным для нас подробностям и игнорировать те подробности, которые нам не интересны. Например, преподаватель — это человек, и он имеет множество атрибутов и вариантов поведения, которые вам интересны. Тем не менее, вы, возможно, игнорируете множество атрибутов и вариантов поведения преподавателя и фокусируетесь только на тех, которые имеют отношение к вашему образованию.

Аналогично, преподаватель фокусируется на ваших атрибутах и вариантах поведения, которые показывают, как хорошо вы изучаете материал на занятиях. Другие атрибуты, такие как ваша работа на других занятиях или ваш рост и вес, игнорируются преподавателем.

И вы, и преподаватель упрощаете ваши отношения, решая, какие атрибуты и варианты поведения являются важными для ваших целей, и используя в ваших отношениях только их.

Несмотря на то, что в различных источниках делается акцент на те или иные особенности внедрения и применения объектно-ориентированного программирования (ООП), 3 основных (базовых) понятия ООП остаются неизменными. К ним относятся:

- наследование (Inheritance),
- инкапсуляция (Encapsulation),
- полиморфизм (Polymorphism).

ООП позволяет разложить проблему на связанные между собой задачи. Каждая проблема становится самостоятельным объектом, содержащим свои собственные коды и данные, которые относятся к этому объекту. В этом случае исходная задача в целом упрощается, и программист получает возможность оперировать с большими по объему программами.

В этом определении ООП отражается известный подход к решению сложных задач, когда мы разбиваем задачу на подзадачи и решаем эти подзадачи по отдельности. С точки зрения программирования подобный подход значительно упрощает разработку и отладку программ.

Центром внимания ООП является объект. Объект - это осязаемая сущность, которая четко проявляет свое поведение.

Объект состоит из следующих трех частей:

- имя объекта;
- состояние (переменные состояния);
- методы (операции).

Объект ООП - это совокупность переменных состояния и связанных с ними методов (операций). Эти методы определяют, как объект взаимодействует с окружающим миром.

Возможность управлять состояниями объекта посредством вызова методов в итоге и определять поведение объекта. Эту совокупность методов часто называют интерфейсом объекта.

Класс (class) - это сложный тип данных, в котором объединены элементы данных (поля) и методы, обрабатывающие эти данные и выполняющие операции по взаимодействию с окружающей средой.

Объект (object) - это конкретная реализация, экземпляр класса. В программировании отношения объекта и класса можно сравнить с описанием переменной, где сама переменная (объект) является экземпляром какого-либо типа данных (класса).

Обычно, если объекты соответствуют конкретным сущностям реального мира, то классы являются некими абстракциями, выступающими в роли понятий. Для формирования какого-либо реального объекта необходимо иметь шаблон, на основании которого и строится создаваемый объект. При рассмотрении основ ООП часто смешивают понятие объекта и класса. Дело в том, что класс - это некоторое абстрактное понятие, а объект - это физическая реализация класса (шаблона).

Методы (methods) - это функции (процедуры), принадлежащие классу. Метод класса вызывается конкретным экземпляром класса и привязан к описанию и структуре класса.

Сообщение (message) - это практически то же самое, что и вызов функций в обычном программировании. В ООП обычно употребляется выражение "послать сообщение" какому-либо объекту. Понятие "сообщение" в ООП можно объяснить с точки зрения основ ООП: мы не можем напрямую изменить состояние объекта и должны как бы послать сообщение объекту, что мы хотим так и так изменить его состояние. Объект сам меняет свое состояние, а мы только его просим об этом, посылая сообщения.

Инкапсуляция - это механизм, который объединяет данные и методы, манипулирующие этими данными, и защищает и то и другое от внешнего вмешательства или неправильного использования. Когда методы и данные объединяются таким способом, создается объект.

Применяя инкапсуляцию, как бы, возводим крепость, которая защищает данные, принадлежащие объекту, от возможных ошибок, которые могут возникнуть при прямом

доступе к этим данным. Кроме того, применение этого принципа очень часто помогает локализовать возможные ошибки в коде программы. Инкапсуляция подразумевает под собой скрытие данных (data hiding), что позволяет защитить эти данные.

Примером применения принципа инкапсуляции являются команды доступа к файлам. Обычно доступ к данным на диске можно осуществить только через специальные функции. Вы не имеете прямой доступ к данным, размещенным на диске. Таким образом, данные, размещенные на диске, можно рассматривать скрытыми от прямого вмешательства. Доступ к ним можно получить с помощью специальных функций, которые по своей роли схожи с методами объектов.

Наследование - это процесс, посредством которого, один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него.

Исследователи во многих областях естествознания тратят львиную долю времени на классификацию объектов в соответствии с определенными особенностями. Для животных, растений существуют специальные принципы классификации и разбиения на классы, подклассы, виды и подвиды и т.д. В результате образуется своего рода иерархия или дерево с одной общей категорией в корне и подкатегориями, разветвляющимися на подкатегории и т.д.

Пытаясь провести классификацию некоторых новых животных или объектов, мы задаем следующие вопросы: В чем сходство этого объекта с другими объектами общего класса? В чем различия?

Каждый класс имеет набор поведений и характеристик, которые его определяют. Более высокие уровни являются более общими. Каждый уровень является более специфическим, чем предыдущий уровень и менее общим. Когда характеристика определена, все категории ниже этого определения включают эту характеристику. Поэтому, когда мы говорим про того или иного конкретного представителя класса (отряда, вида и т.д.), то нам не надо говорить про его общие особенности, характерные для этого класса, а говорим только про его специфические особенности в рамках этого класса.

Смысл и универсальность наследования заключается в том, что не надо каждый раз заново (с нуля) описывать новый объект, а можно указать родителя (базовый класс) и описать отличительные особенности нового класса. В результате, новый объект будет обладать всеми свойствами родительского класса плюс своими собственными отличительными особенностями.

Пример: Создадим базовый класс "транспортное средство", который универсален для всех средств передвижения на 4-х колесах. Этот класс "знает" как двигаются колеса, как они поворачивают, тормозят и т.д. А затем на основе этого класса создадим класс "легковой автомобиль", который унаследуем из класса "транспортное средство". Поскольку новый класс является наследником класса "транспортное средство", то класс "легковой автомобиль" унаследовал все особенности класса "транспортное средство" и в этом случае не надо в очередной раз описывать как двигаются колеса и т.д. После построения класса "легковой автомобиль" можно добавить особенности поведения, которые характерны для легковых автомобилей. В то же время можем взять за основу этот же класс "транспортное средство" и построить класса "грузовые автомобили". Описав отличительные особенности грузовых автомобилей, получим уже новый класс "грузовые автомобили". А, к примеру, на основании класса "грузовой автомобиль" уже можно описать определенный подкласс грузовиков и т.д. Таким образом, не надо каждый раз описывать все "с нуля". В этом и заключается главное преимущество использования механизма наследования. Сначала формируем простой шаблон, а затем все усложняя и конкретизируя, поэтапно создаем все более сложный шаблон.

В данном случае был приведен пример простого наследования, когда наследование производится только из одного класса. В некоторых объектно-ориентированных языках программирования определены механизмы наследования, позволяющие наследовать из одного и более класса. Однако реализация подобных механизмов зависят от самого применяемого языка ООП. Кроме того, следует отметить, что особенности реализации даже простого наследования могут различаться от языка ООП к языку.

В описаниях языков ООП принято класс, из которого наследуют называть родительским классом (parent class) или основой класса (base class). Класс, который получаем в результате наследования, называется порожденным классом (derived or child class). Родительский класс всегда считается более общим и развернутым. Порожденный же класс всегда более строгий и конкретный, что делает его более удобным в применении при конкретной реализации.

ООП - это процесс построения иерархии классов. А одним из наиболее важных свойств ООП является механизм, по которому типы классов могут наследовать характеристики из более простых, общих типов. Этот механизм называется наследованием. Наследование обеспечивает общность функций, в то же время допуская столько особенностей, сколько необходимо.

Полиморфизм – это свойство, которое позволяет одно и тоже имя использовать для решения нескольких технически разных задач.

В общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. Выбор конкретного действия, в зависимости от ситуации, возлагается на компилятор.

Применительно к ООП, целью полиморфизма, является использование одного имени для задания общих для класса действий. На практике это означает способность объектов выбирать внутреннюю процедуру (метод) исходя из типа данных, принятых в сообщении.

Представьте, что нужно открыть замок и у нас есть связка ключей. Для того, чтобы открыть дверь мы перебираем один ключ за другим пока не найдем подходящий. Т.е. когда шаблон замка совпадает с шаблоном параметров ключа, замок открывается. Аналогично работает компилятор при наличие нескольких функций. Он последовательно проверяет шаблоны функций с одним и тем же именем пока не найдет подходящий.

Механизм работы ООП можно описать примерно так: при вызове того или иного метода класса сначала ищется метод у самого класса. Если метод найден, то он выполняется и поиск этого метода на этом завершается. Если же метод не найден, то обращаемся к родительскому классу и ищем вызванный метод у него. Если найден – поступаем как при нахождении метода в самом классе. А если нет - продолжаем дальнейший поиск вверх по иерархическому дереву. Вплоть до корня (верхнего класса) иерархии.

2.2 Синтаксис объявления класса

Классы C# – это шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными. Класс определяет, какие данные и какую функциональность может иметь каждый конкретный объект (экземпляр) этого класса. Например, класс, представляющий студента, может определять такие поля, как studentID, firstName, lastName, group, и т.д. которые нужны для хранения информации о конкретном студенте.

Класс также может определять функциональность, которая работает с данными, хранящимися в этих полях. Вы создаете экземпляр этого класса для представления конкретного студента, устанавливаете значения полей экземпляра и используете его функциональность. При создании классов, как и всех ссылочных типов – используется ключевое слово new для выделения памяти под экземпляр. В результате объект создастся и инициализируется (числовые поля инициализируются нулями, логические – false, ссылочные – в null и т.д.).

Синтаксис объявления и инициализации класса:

[спецификатор][модификатор] Class <имя класса>
{

```
        [спецификатор][модификатор]   тип <имя поля1>;
        [спецификатор][модификатор]   тип <имя поля2>;
        …
        [спецификатор][модификатор]   тип <Метод1()>
        {...}
        [спецификатор][модификатор]   тип <Метод2()>
        {...}
}
```

Пример объявления класса описывающего студента:

```
class Student
{
        public int studentID;
        public string firstName;
        public string lastName;
        public string group;
}

class Program
{
        static void Main(string[] args)
        {
                Student st1;
                st1 = new Student();
                Student st2 = new Student();
        }
}
```

Доступ к полям и методам класса осуществляется через «.» из- под объекта класса. Доступ к содержимому класса вне его границ, осуществляется только к общедоступным данным. Остальные остаются инкапсулированными.

2.3 Спецификаторы доступа

С помощью спецификаторов доступа можно регулировать доступность некоторого типа или данных внутри типа.

При определении класса с видимостью в рамках файла, а не другого класса, его можно сделать открытым (**public**) или внутренним (**internal**). Открытый тип доступен любому коду любой сборки. Внутренний класс доступен только из сборки, где он определен. По умолчанию компилятор C# делает класс внутренним.

При определении члена класса (в том числе вложенного) можно указать спецификатор доступа к члену. Спецификаторы определяют, на какие члены можно ссылаться из кода. В общеязыковой среде выполнения (Common Language Runtime, CLR) определен свой набор возможных спецификаторов доступа, но в каждом языке программирования существует свой синтаксис и термины. Рассмотрим спецификаторы определяющие уровень ограничения – от максимального (**private**) до минимального (**public**):

private – данные доступны только методам внутри класса и вложенных в него классам,

protected – данные доступны только методам внутри класса (и вложенным в него классам) или одном из его производных классов,

internal – данные доступны только методам в сборке

protected internal – данные доступны только методам вложенного или производного типа класса и любым методам сборки,

public – данные доступны всем методам во всех сборках.

Доступ к члену класса можно получить, только если он определен в видимом классе. То есть, если в сборке А определен внутренний класс, имеющий открытый метод, то код сборки Б не сможет вызвать открытый метод, поскольку внутренний класс сборки А не доступен из Б.

В процессе компиляции кода компилятор проверяет корректность обращения кода к классам и членам. Обнаружив некорректную ссылку на какие-либо классы или члены, выдается ошибка компиляции.

Если не указать явно спецификатор доступа, компилятор C# выберет по умолчанию закрытый – наиболее строгий из всех.

Если в производном классе переопределяется член базового – компилятор C# потребует, чтобы у членов базового и производного классов был одинаковый спецификатор доступа. При наследовании базовому классу CLR позволяет понижать, но не повышать уровень доступа к члену.

2.4 Поля класса

Поле – это переменная, которая хранит значение любого стандартного типа или ссылку на ссылочный тип. При объявлении полей могут указываться следующие модификаторы:

- Если модификатор не указывать, то это означает, что поле связано с экземпляром класса, а не самим классом.
- Модификатор **static** – означает, что поле является частью класса, а не объекта.
- Модификатор **readonly** – означает, что поле будет использоваться только для чтения и запись в поле разрешается только из кода метода конструктора либо сразу при объявлении.

CLR поддерживает изменяемые (**read/write**) и неизменяемые (**readonly**) поля. Большинство полей – изменяемые. Это означает, что значение таких полей может многократно меняться во время исполнения кода. Неизменяемые поля сродни константам, но являются более гибкими, так как значение константам можно задать только при объявлении, а у неизменяемых полей это еще можно сделать и в конструкторах. Важно понимать, что неизменность поля ссылочного типа означает неизменность ссылки, которую оно содержит, но только не объекта, на которую эта ссылка указывает. То есть перенаправить ссылку на другое место в памяти мы не можем, но изменить значение объекта, на который указывает ссылка – можем. Значения неизменяемых полей значимых типов – изменять не можем.

Пример:

```
class MyClass
{
        public readonly int var1 = 10;
        public readonly int[] myArr = { 1, 2, 3 };
        public readonly int var2;       // инициализация readonly
                                                // поля при объявлении
        public MyClass(int i)
        {
                var2 = i;       // инициализация readonly
        }                       // поля в конструкторе
}

class Program
{
        static void Main(string[] args)
        {
                MyClass obj = new MyClass();
                obj.var = 100;                          // Ошибка
```

```
            obj.myArr = new int[10];          // Ошибка
            obj.myArr[0] = 11;                //Ошибки нет
        }
}
```

Если объявляется статическое поле, то оно принадлежит классу в целом, а не конкретному объекту. И соответственно получить доступ к такому объекту можно только из-под класса, используя следующий синтаксис:

<Имя класса>.<имя поля>

Например, есть класс Bank. В этом классе будет статическое поле balance. Сымитируем ситуацию, когда в любом филиале банка можно будет положить деньги на депозит или взять кредит. Пусть все филиалы работают с общим счетом.

```
class Bank
{
        public static float balance = 1000000;
}
class Program
{
        static void Main(string[] args)
        {
                Bank filial1 = new Bank();
                Bank filial2 = new Bank();
                Console.WriteLine("1-ому филиалу доступно {0:C}",
                                Bank.balance);
                Console.WriteLine("2-ому филиалу доступно {0:C}",
                                Bank.balance);
                Console.WriteLine("В 1-ом филиале взяли кредит на
                100000" + ", осталось {0:C}", Bank.balance-=100000);
                Console.WriteLine("2-ому филиалу доступно {0:C}",
                                Bank.balance);
                Console.WriteLine("В 2-ом филиале взяли кредит на
                200000" +", осталось {0:C}", Bank.balance -= 200000);
                Console.WriteLine("1-ому филиалу доступно {0:C}",
                                Bank.balance);
                Console.WriteLine("В 1-ом филиале открыли депозит на "
                + "200000, осталось {0:C}", Bank.balance += 200000);
        }
}
```

Результат выполнения программы изображен на рисунке 2.2.

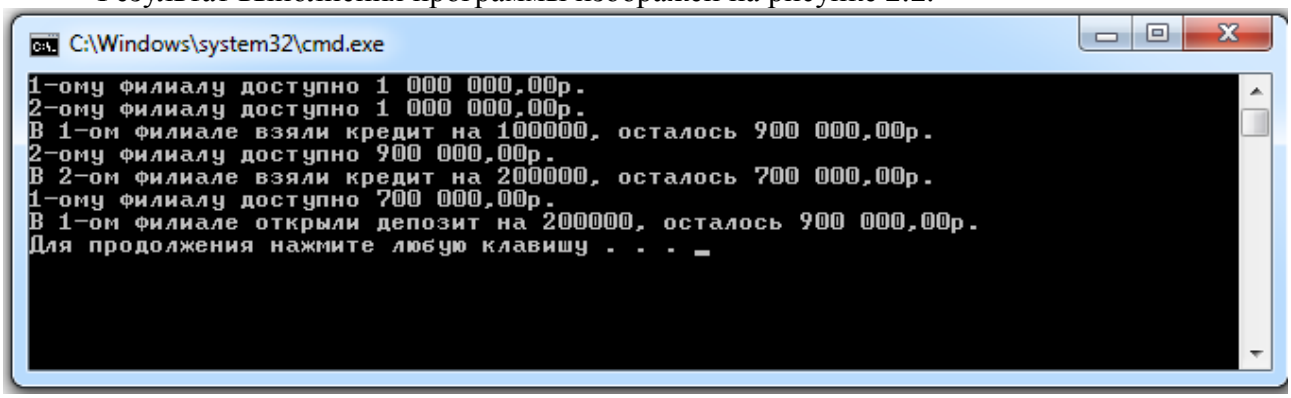

Рис. 2.2 Результат выполнения программы.

2.5 Методы класса

Описание метода

Все функции в языке C# обязательно должны определяться внутри классов или структур. Каждый метод должен быть объявлен отдельно как общедоступный или приватный. То есть блоки public:, private: для группировки нескольких методов использовать нельзя. Все методы C# объявляются и определяются внутри определения класса, таким образом нельзя отделить реализацию метода от объявления.

Определение метода состоит из: спецификаторов и модификаторов, типа возвращаемого значения, за которым следует имя метода, списка аргументов (если они есть), заключенных в скобки, и тела метода, заключенного в фигурные скобки:

[спецификаторы][модификаторы] тип_возврата
<Имя Метода>([параметры])
{
// Тело метода
}

Добавим в класс Student (описанный выше) метод. При этом сделаем все поля класса закрытыми. Используя практику сокрытия данных класса и предоставления открытых методов по работе с этими данными, тем самым поддерживается принцип инкапсуляции данных и уменьшается вероятность некорректной работы программы.

```
class Student
{
	private string firstName = "Петя";
	public void ShowName()
	{
		Console.WriteLine(firstName);
	}
}
class Program
{
	static void Main(string[] args)
	{
		Student st = new Student();
		st.ShowName();
	}
}
```

Результат выполнения программы изображен на рисунке 2.3.

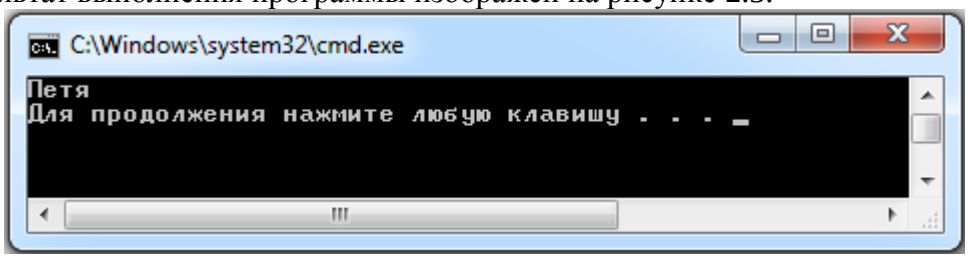

Рис. 2.3 Результат выполнения программы.

Для работы со статическими полями – были введены статические методы. Эти методы, как и статические поля, принадлежат классу, а не объекту. Они исключают возможность вызова из-под объекта и соответственно не работают с нестатическими полями.

Предположим, что все создаваемые нами студенты будут студентами ТюмГУ и добавим в наш класс Student статическое поле, которое будет содержать имя учебного

заведения. Чтоб поле нельзя было изменить – сделаем его закрытым и позволим статическому методу ShowUniversity работать с нашим полем в режиме чтения.

```
class Student
  {
    private static string UniversityName = "ТюмГУ";
    // старые поля и методы остаются без изменения
    public static void ShowUniversity()
    { Console.WriteLine(UniversityName); }
  }
  class Program
  {
    static void Main(string[] args)
    { Student.ShowUniversity(); }
  }
```

Результат выполнения программы изображен на рисунке 2.4.

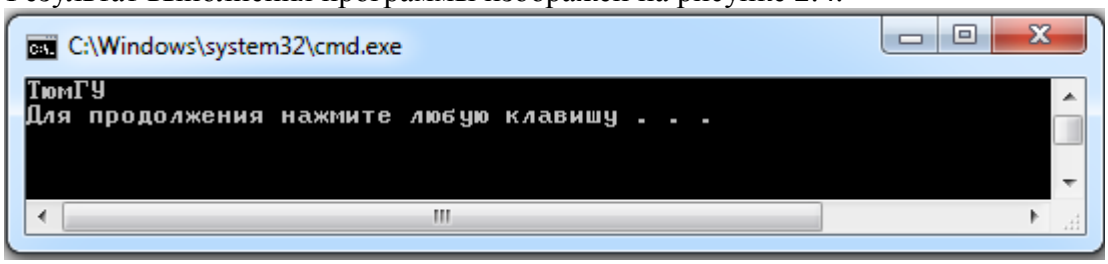

Рис. 2.4 Результат выполнения программы.

Передача параметров

При вызове метода выполняются следующие действия:

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода в соответствии с их типом.
3. Каждому из параметров сопоставляется соответствующий аргумент.
4. Выполняется тело метода.
5. Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип **void** , управление передается на оператор, следующий после вызова.

При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение.

Параметры, указываемые в заголовке метода при его описании, называются формальными параметрами.

Параметры, указываемые при вызове метода, называются фактическими параметрами.

Корректность передачи параметров гарантируется соблюдением порядка перечисления в заголовке метода и совместимостью по присваиванию между соответствующими фактическими и формальными параметрами.

Существуют два способа передачи параметров: по значению и по ссылке. Когда переменная передастся по ссылке, вызываемый метод получает саму переменную, поэтому любые изменения, которым производятся над переменной внутри метода, останутся в силе после его завершения. С другой стороны, если переменная передается по значению, то вызываемый метод получает копию этой переменной, и соответственно все изменения в копии по завершении метода будут утеряны. Для сложных типов данных передача по ссылке более эффективна из-за большого объема данных, который приходится копировать при передаче по значению.

При передаче параметров нужно быть осторожным в отношении ссылочных типов. Поскольку переменная ссылочного типа содержит лишь ссылку на объект, то именно ссылка будет скопирована при передаче параметра, а не сам объект. То есть изменения,

произведенные в самом объекте, сохранятся. В отличие от этого переменные типа значений действительно содержат данные, поэтому в метод передается копия самих данных. Например, переменная a передастся в метод по значению, и любые изменения, которые сделает метод с соответствующим переменной a формальным параметром, не изменят значения переменной a. В противоположность этому, если в метод передастся массив или любой другой ссылочный тип, такой как класс, то изменения, сделанные при выполнении тела метода, будут отражены в исходном объекте массива.

Пример:

```
class Program
{
        static void MyFunctionByVal(int[] myArr, int i)
        { myArr[0] = 100;  i = 100; }
        static void Main(string[] args)
        {
                int i = 0;
                int[] myArr = { 0, 1, 2, 4 };
                Console.WriteLine("i = {0}", i);
                Console.WriteLine("MyArr[0] = {0}", myArr[0]);
                Console.WriteLine("Вызов MyFunction");
                MyFunctionByVal (myArr, i);
                Console.WriteLine("i = {0}", i);
                Console.WriteLine("MyArr[0] = {0}", myArr[0]);
        }
}
```

Результат выполнения программы изображен на рисунке 2.5.

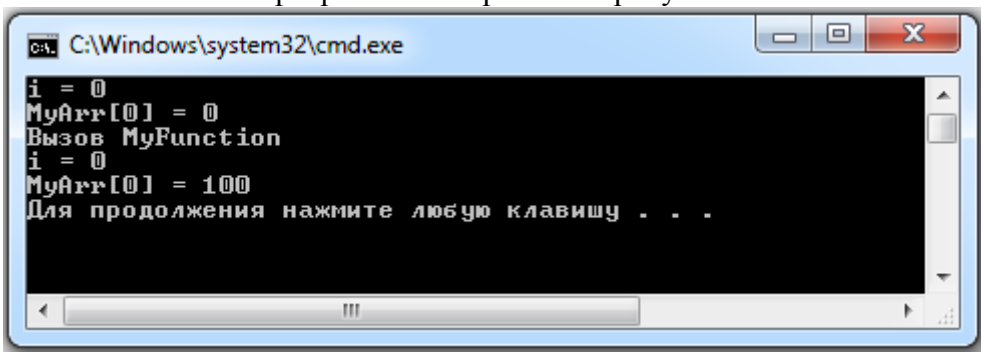

Рис. 2.5 Результат выполнения программы.

Передача параметров по значению.

При вызове метода:

а) выполняется выделение памяти под формальные параметры и локальные данные (в стеке или в специальной области памяти для локальных данных);

б) выполняется копирование значений фактических параметров в память, выделенную для формальных параметров.

Во время выполнения метода:

а) изменение значений формальных параметров не оказывает никакого влияния на содержимое ячеек памяти фактических параметров.

При окончании выполнения метода:

а) память, выделенная под формальные параметры и локальные данные, очищается;

б) новые значения формальных параметров, полученные в процессе выполнения тела метода, теряются вместе с очисткой памяти.

Параметр-значение описывается в заголовке метода следующим образом:

тип имя

Пример:

```
class Program
   {
     private static void MyFunction(int i)
      {
         Console.WriteLine("Внутри функции MyFunction до
                изменения i = {0}", i);
      i = 100;
      Console.WriteLine("Внутри функции MyFunction после
             изменения i = {0}", i);
      }
     static void Main(string[] args)
      {
      int i = 10;
      Console.WriteLine("Внутри метода Main до передачи в метод
                MyFunction i = {0}", i);
      MyFunction(i);
      Console.WriteLine("Внутри метода Main после передачи в метод MyFunction i =
{0}", i);
      }
   }
```

Результат выполнения программы изображен на рисунке 2.6.

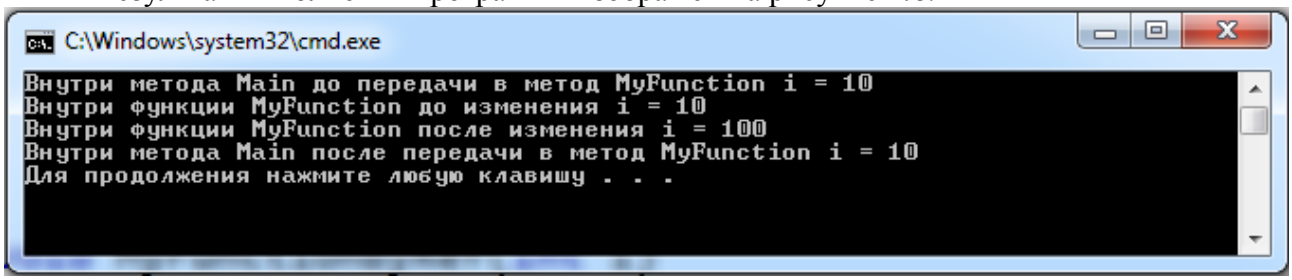

Рис. 2.6 Результат выполнения программы.

Передача параметров по ссылке (адресу).

При вызове метода:

а) выполняется выделение памяти только для локальных данных и для сохранения адресов фактических параметров (в стеке или в специальной области памяти для локальных данных);

б) выполняется копирование адресов (но не значений!) фактических параметров в выделенную память для локальных параметров;

в) использовать в качестве фактических параметров константы запрещено.

Во время выполнения метода:

а) никаких ограничений на использование параметров данного вида не накладывается;

б) изменение значений формальных параметров, используя скопированные адреса, выполняется непосредственно на ячейках памяти соответствующих фактических параметров.

При окончании выполнения метода:

а) специального копирования результата не требуется, поскольку все действия с формальными параметрами выполнялись непосредственно над ячейками памяти фактических параметров;

б) память, выделенная для работы метода, очищается.

Использование модификатора ref.

Ключевым словом ref помечаются те параметры, которые должны передаваться в метод по ссылке. Таким образом, мы будем внутри метода манипулировать данными, объявленными в вызывающем методе. Аргументы, которые передаются в метод с ключевым словом ref,

обязательно должны быть проинициализированы, иначе компилятор выдаст сообщение об ошибке.

Описание параметра-ссылки в заголовке метода следующее:

ref тип имя

Пример:

```
class Program
   {
     private static void MyFunctionByRef(ref int i)
      {
         Console.WriteLine("Внутри функции MyFunction до
                изменения i = {0}", i);
       i = 100;
       Console.WriteLine("Внутри функции MyFunction после
              изменения i = {0}", i);
      }
     static void Main(string[] args)
      {
       int i = 10;
       Console.WriteLine("Внутри метода Main до передачи в метод
              MyFunction i = {0}", i);
       MyFunctionByRef(ref i);
       Console.WriteLine("Внутри метода Main после передачи в
              метод MyFunction i = {0}", i);
      }
   }
```

Результат выполнения программы изображен на рисунке 2.7.

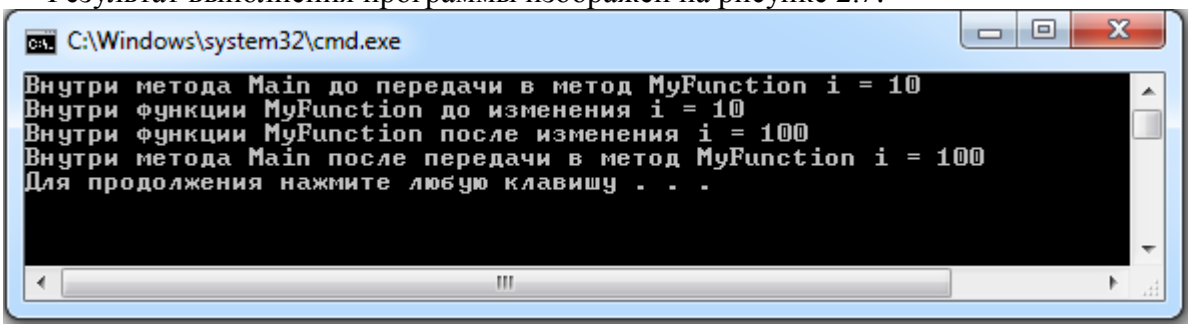

Рис. 2.7 Результат выполнения программы.

Использование модификатора out

Параметры, обозначенные ключевым словом out, также используются для передачи по ссылке. Отличие от ref состоит в том, что параметр считается выходным и соответственно компилятор разрешит не инициализировать его до передачи в метод и проследит, чтоб метод занес в этот параметр значение (иначе будет выдано сообщение об ошибке).

Описание параметра-ссылки в заголовке метода следующее:

out тип имя

Пример:

```
class Program
{
        static void Main(string[] args)
        {
                int i;
                GetDigit(out i);
                Console.WriteLine("i = " + i);
        }
        private static void GetDigit(out int digit)
```

```
        {
            digit = new Random().Next(10);
        }
}
```

Результат выполнения программы изображен на рисунке 2.8.

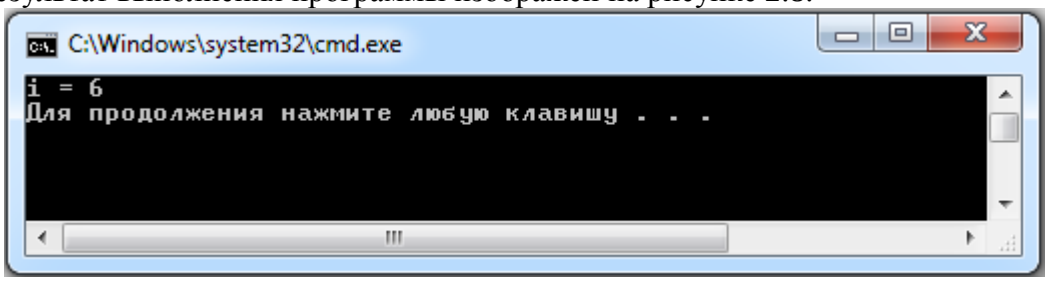

Рис. 2.8 Результат выполнения программы.

Создание методов с переменным количеством параметров

Иногда бывает удобно создать метод, в который можно передавать разное количество аргументов. Язык C# предоставляет такую возможность с помощью ключевого слова **params**. Параметр, помеченный этим ключевым словом, размещается в списке параметров последним и обозначает массив заданного типа неопределенной длины, например:

public int Sum(int a, out int b. params int [ ] c) ...

В этот метод можно передать три и более параметров. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество элементов массива получают с помощью его свойства **Length**. При использовании ключевого слова необходимо учитывать, что параметр, помечаемый ключевым словом **params** должен стоять последним в списке параметров.

Пример:

```
class Program
{
        static void Main(string[] args)
        {
            Console.WriteLine("Сумма = " + Sum( 1, 2, 3, 4, 5 ));
        }
        private static int Sum(params int[] arr)
        {
            int res = 0;
            foreach (int i in arr)
            res += i;
            return res;
        }
}
```

Результат выполнения программы изображен на рисунке 2.9.

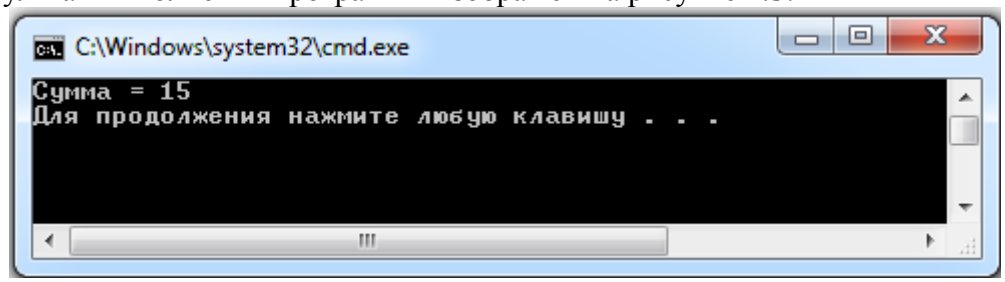

Рис. 2.9 Результат выполнения программы.

Ключевое слово return

Метод может завершить свое выполнение тремя способами:

- Когда управление дойдет до завершающей фигурной скобки (при этом метод ничего не возвращает).
- Когда управление дойдет до ключевого слова **return** (без возвращаемого значения и тип возвращаемого значения метода – **void**).
- Когда управление дойдет до ключевого слова **return** (после которого стоит возвращаемое значение, метод что-либо возвращает).

Метод может содержать несколько операторов **return**, если это необходимо для реализации алгоритма. Если метод описан как **void**, выражение не указывается. Выражение, указанное после **return**, неявно преобразуется к типу возвращаемого методом значения и передается в точку вызова метода.

```
private static int f1() { return 1; } // правильно
private static void f2() { return 1; } // неправильно, f2 не должен возвращать значение
private static double f3() { return 1; } // правильно, 1 неявно преобразуется к типу double
```

Метод, возвращающий значение, должен делать это с помощью оператора **return**. Если поток управления достигнет конца метода, возвращающего значение, не встретив на своем пути оператор **return**, в вызывающий модуль вернется "мусор".

Если метод не объявлен со спецификатором **void**, его можно использовать в качестве операнда в любом выражении.

Пример: Программа возвращает позицию числа в неотсортированном массиве, а если число не найдено, возвращает число –1.

```
class Program
  {
    static void Main(string[] args)
    {
      int num;
      Console.WriteLine("Введите искомое число: ");
      string buf = Console.ReadLine();
      int x = Convert.ToInt32(buf);
      if ((num = find_number(x)) == -1)
         Console.WriteLine("Такое число не найдено");
      else Console.WriteLine("Позиция числа в массиве {0}", num);

    }
    private static int find_number(int number)
     {
      int N = 7, i;
      int[] array = { 21, 2, -4, 6, 1, 11, -3 };
      for (i = 0; i < N; i++)
        if (number == array[i]) return i;
      return -1;
     }
  }
```

Результат выполнения программы изображен на рисунке 2.10.

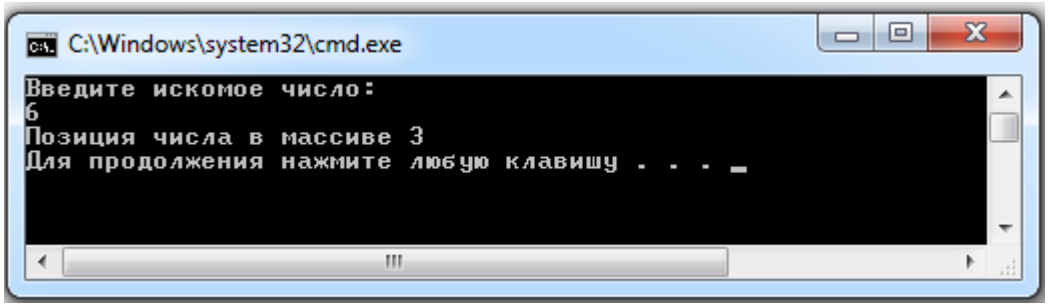

Рис. 2.10 Результат выполнения программы.

Методы можно разделить на три категории:

К первой категории относятся вычислительные методы. Они выполняют некие операции над своими аргументами и возвращают результат вычислений.

Методы второго типа выполняют обработку информации. Их возвращаемое значение просто означает, успешно ли завершены операции.

Методы третьего вида не имеют явно возвращаемого значения. По существу, эти методы являются процедурами и не вычисляют никакого результата.

Пример:

```
class Program
   {
      static void Main(string[] args)
      {
         int x, y, z;
         x = 10; y = 20;
         z = mul(x, y);                    /* 1 */
         Console.WriteLine(mul(x,y));   /* 2 */
         mul(x, y);                        /* 3 */
       }

      private static int mul(int a, int b)
       {
         return a * b;
       }
    }
```

В первой строке значение, возвращаемое методом **mul()**, присваивается переменной z. Во второй строке возвращаемое значение ничему не присваивается, но используется внутри метода **Console.WriteLine().** В третьей строке значение, возвращаемое с помощью оператора **return**, отбрасывается.

Рекурсивные методы

Рекурсивным называется метод, который вызывает сам себя. Такая рекурсия называется прямой. Существует косвенная рекурсия, когда два или более метода вызывают друг друга. Если метод вызывает себя, в стеке создается копия значений его параметров, после чего управление передается первому исполняемому оператору метода. При повторном вызове этот процесс повторяется. Для завершения вычислений каждый рекурсивный метод должен содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении метода соответствующая часть стека освобождается, и управление передается вызывающему методу, выполнение которого продолжается с точки, следующей за рекурсивным вызовом.

Пример: Вычислить значение факториала.

Функцию факториала n! определяют как произведение первых n целых чисел:

n! = 1 * 2 * 3 * … * n

Такое произведение легко вычислить с помощью итеративных конструкций (например for).

Другое определение факториала, в котором используется рекуррентная формула, имеет вид:

(1) 0!=1

(2) для ∀ n>0 n!=n*(n-1)!

```
private static int fact ( int n ) // описание рекурсивного метода
        {
         if ( n <= 1 ) return 1 ; // нерекурсивная ветвь
         return ( n*fact (n - 1)) ; // метод вызывает сам себя
        }
```

Или

```
private static int fact ( int n ) // описание рекурсивного метода
       {
        return ( n > 1 ) ? n*fact ( n - 1 ) : 1 ;
       }
```

Пример: Для заданного числа вычислить число Фибоначчи.

Для чисел Фибоначчи рекурсивное определение выглядит следующим образом:

(1) F(1)=1,

(2) F(2)=1,

(3) для ∀ n>2 F(n)=F(n-1)+F(n-2)

```
class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Введите искомое число: ");
         string buf = Console.ReadLine();
         int n = Convert.ToInt32(buf);
         Console.WriteLine("Число Фибоначчи = {0}", Fib(n));
       }
      private static int Fib(int i)
      {
         if ((i == 1) || (i == 2)) return 1;
         else return Fib(i - 1) + Fib(i - 2);
      }
    }
}
```

Перегрузка методов

В C# несколько методов могут иметь одно и то же имя. В этом случае метод, идентифицируемый этим именем, называется перегруженным. Перегрузить можно только методы, которые отличаются либо типом, либо числом своих аргументов. Перегрузить методы, которые отличаются только типом возвращаемого значения, нельзя. Перегружаемые методы дают возможность упростить программы, допуская обращение к одному имени для выполнения близких по смыслу действий.

Чтобы перегрузить некоторый метод, нужно объявить, а затем определить все требуемые варианты его вызова. Компилятор автоматически выберет правильный вариант вызова на основании числа и типа используемых аргументов.

На перегруженные методы накладываются несколько ограничений:

- любые два перегруженные методы должны иметь различные списки параметров;
- перегрузка методов с совпадающими списками аргументов на основе лишь типа возвращаемых ими значений недопустима;

- методы не могут быть перегружены исключительно на основе того, что один из них является статическим, а другой – нет;
- типы "массив" и "указатель" рассматриваются как идентичные с точки зрения перегрузки.

Пример:

```
class Program
  {
    static void Main(string[] args)
    {
       int mN, N = -255; float mF, F = -25.0f; double mD, D = -2.55;
       mN = abs(N); // вызов перегруженной функции abs ( int )
       mF = abs(F); // вызов перегруженной функции abs ( float )
       mD = abs(D); // вызов перегруженной функции abs ( double )
       Console.WriteLine("abs(int) \t| {0}\t| = {1}", N, mN);
       Console.WriteLine("abs(float) \t| {0}\t| = {1}", F, mF);
       Console.WriteLine("abs(double) \t| {0}\t| = {1}", D, mD);
    }
    public static int abs(int a)
      { return a < 0 ? -a : a; }
    public static float abs(float a)
      { return a < 0 ? -a : a; }
    public static double abs(double a)
      { return a < 0 ? -a : a; }
  }
```

Результат выполнения программы изображен на рисунке 2.11.

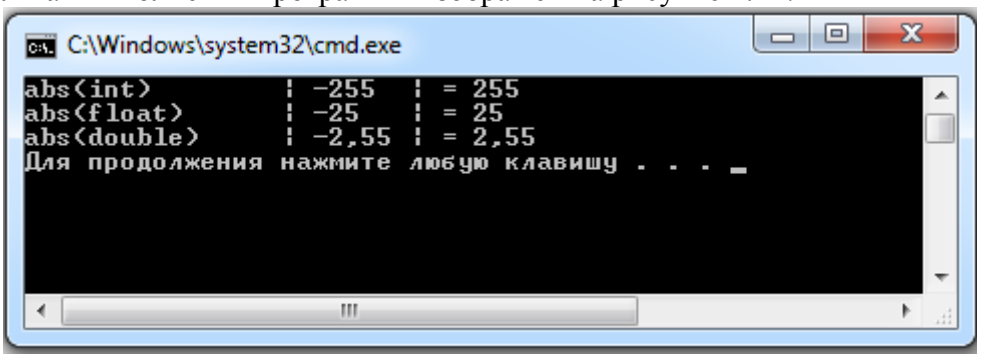

Рис. 2.11 Результат выполнения программы.

2.6 Ключевое слово this

Каждый объект содержит свой экземпляр полей класса. Методы находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр **this**, в котором хранится ссылка на вызвавший функцию экземпляр.

В явном виде параметр **this** применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

```
class Demo
  {
    double y;
    public Demo T() // метод возвращает ссылку на экземпляр
      { return this; }
```

```
    public void Sety(double y)
      { this.y = y; } // полю y присваивается значение параметра y
  }
```

2.7 Конструкторы

Понятие конструктора

Конструкторы – это методы класса, которые вызываются при создании объекта.

Конструктор предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции **new**. Имя конструктора совпадает с именем класса. Конструкторы обладают следующими свойствами:

- Конструктор не возвращает значение, даже типа **void.**
- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.
- Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов — значение **null.**
- Конструктор, вызываемый без параметров, называется ***конструктором по умолчанию.***

В C# существует 3 вида конструкторов:

- Конструктор по умолчанию.
- Конструктор с параметрами – конструктор, который может принимать необходимое количество параметров для инициализации полей класса или каких-либо других действий.
- Статический конструктор – конструктор, относящийся к классу, а не к объекту. Существует для инициализации статических полей класса. Определяется без какого-либо спецификатора доступа с ключевым словом **static.**

При создании конструкторов нужно помнить, что все конструкторы (кроме статического) имеют спецификатор доступа **public**, все конструкторы, кроме конструктора по умолчанию и статического конструктора, могут иметь необходимое количество параметров. Конструктор по умолчанию может быть только один. Если конструктор описан в классе, то конструктор по умолчанию, который вам предоставлялся компилятором, работать не будет.

Пример определения своего конструктора по умолчанию. Для работы с конструкторами создан новый класс, описывающий машину.

```
class Car
  {
    private string driverName; // Имя водителя
    private int currSpeed; // Текущая скорость
    public Car() // Конструктор по умолчанию
    {
      driverName = "Михаель Шумахер";
      currSpeed = 10;
    }
    public void PrintState() // Распечатка текущих данных
    {
      Console.WriteLine("{0} едет со скоростью {1} км/ч.",
      driverName, currSpeed);
    }
    public void SpeedUp(int delta) // Увеличение скорости
    { currSpeed += delta; }
  }
  class Program
```

```
    {
        static void Main(string[] args)
        {
            Car myCar = new Car();
            for (int i = 0; i <= 10; i++)
            {
                myCar.SpeedUp(5);
                myCar.PrintState();
            }
        }
    }
```

Результат выполнения программы изображен на рисунке 2.12.

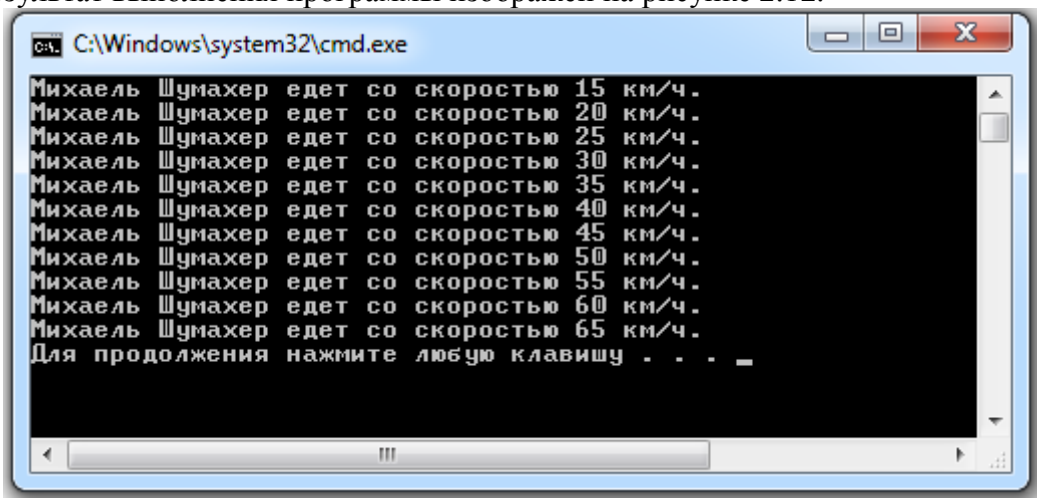

Рис. 2.12 Результат выполнения программы.

Конструктор с параметрами

Конструктор с параметрами отличается от конструктора по умолчанию наличием параметров. Количество параметров определяет количество полей, которые необходимо проинициализировать при создании объекта.

Пример: Добавить в класс Car конструктор позволяющий инициализировать поле driverName.

```
class Car
{
        // Старые поля и методы...
        public Car(string name)
        {
                driverName = name;
                currSpeed = 10;
        }
}
class Program
{
        static void Main(string[] args)
        {
                Car myCar = new Car("Рубенс Барикелло");
                for (int i = 0; i <= 10; i++)
                {
                        myCar.SpeedUp(5);
                        myCar.PrintState();
                }
```

```
        }
}
```

Перегруженные конструкторы

Конструкторы, как и другие методы, можно перегружать: создавать в классе несколько конструкторов с различными списками параметров, чтобы обеспечить возможность инициализации объектов разными способами.

Пример: определим конструкторы с параметрами для класса Car.

```
class Car
{
        // Старые поля и методы...
        public Car(string name)
        { driverName = name; currSpeed = 10; }
        public Car(string name, int speed)
        { driverName = name; currSpeed = speed; }
}
class Program
{
        static void Main(string[] args)
        {
                Car myCar = new Car("Ральф Шумахер", 10);
                for (int i = 0; i <= 10; i++)
                {
                myCar.SpeedUp(5);
                myCar.PrintState();
                }
        }
}
```

Статические конструкторы

Статический конструктор связан с классом, а не с конкретным объектом. Сам конструктор нужен для инициализации статических данных. Когда вызывается этот конструктор – неизвестно, но гарантируется, что вызов произойдет до первого создания объекта класса. Для примера со статическим конструктором создадим класс описывающий банковские филиалы, но на этот раз статическое поле будет содержать бонус в процентах для оформления депозитов. А текущий баланс у каждого филиала будет свой.

```
class Bank
   {
      private double currBalance;
      private static double bonus;
      public Bank(double balance)
      { currBalance = balance; }
      static Bank()
      { bonus = 1.04; }
      public static void SetBonus(double newRate)
      { bonus = newRate; }
      public static double GetBonus()
      {  return bonus; }
      public double GetPercents(double summa)
      {
         if ((currBalance - summa) > 0)
         {
```

```
            double percent = summa * bonus;
            currBalance -= percent;
            return percent;
        }
        return -1;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Bank b1 = new Bank(1000000);
        Console.WriteLine("Текущий бонусный процент: " +
        Bank.GetBonus());
        Console.WriteLine("Ваш депозит на {0:C}, в кассе забрать:" +
        "{1:C}", 10000, b1.GetPercents(10000));
    }
}
```

Результат выполнения программы изображен на рисунке 2.13.

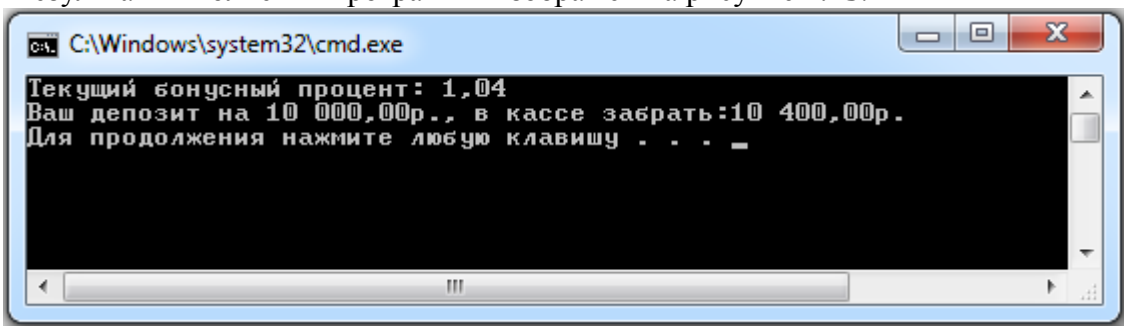

Рис. 2.13 Результат выполнения программы.

3 Контрольные вопросы

1. Что такое объект?
2. Что такое атрибут объекта?
3. Что такое поведение объекта?
4. В чем заключается принцип объектно-ориентированного программирования?
5. Почему важно использовать объектно-ориентированное программирование при разработке компьютерных систем?
6. Каким образом объектно-ориентированное программирование помогает поддерживать сложные компьютерные системы?
7. Определите атрибуты объекта Форма заказа на рис. 2.1.
8. Определите варианты поведения объекта Форма заказа на рис. 2.1.
9. Что такое метод?
10. Объясните роль наследования в объектно-ориентированном программировании.
11. Объясните роль инкапсуляции в объектно-ориентированном программировании.
12. Объясните понятие инкапсуляции в объектно-ориентированном программировании.
13. Что понимается под термином «класс»?
14. Какие элементы определяются в составе класса?
15. Каково соотношение понятий «класс» и «объект»?
16. Что понимается под термином «члены класса»?
17. Какие члены класса Вам известны?

18. Какие члены класса содержат код?
19. Какие члены класса содержат данные?
20. Перечислите пять разновидностей членов класса специфичных для языка C#.
21. Что понимается под термином «конструктор»?
22. Сколько конструкторов может содержать класс?
23. Приведите синтаксис описания класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
24. Какие модификаторы типа доступа Вам известны?
25. В чем заключаются особенности доступа членов класса с модификатором public?
26. В чем заключаются особенности доступа членов класса с модификатором private?
27. В чем заключаются особенности доступа членов класса с модификатором protected?
28. В чем заключаются особенности доступа членов класса с модификатором internal?
29. Какое ключевое слово языка C# используется при создании объекта?
30. Приведите синтаксис создания объекта в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
31. В чем состоит назначение конструктора?
32. Каждый ли класс языка C# имеет конструктор?
33. Какие умолчания для конструкторов приняты в языке C#?
34. Каким значением инициализируются по умолчанию значения ссылочного типа?
35. В каком случае конструктор по умолчанию не используется?
36. Приведите синтаксис конструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
37. Что понимается под термином «деструктор»?
38. В чем состоит назначение деструктора?
39. Приведите синтаксис деструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
40. Что такое рекурсия?
41. Что такое статистический метод?

4 Задание

1. Написать программу в соответствии с вариантом задания. Для класса предусмотреть конструктор по умолчанию, несколько конструкторов по с параметрами, деструктор, методы: изменения, отображения полей класса и методы согласно задания. Для хранения объектов класса использовать динамический массив или стандартный список List. Описание класса и методов класса должны находиться в отдельном модуле.
2. Отладить и протестировать программу.
3. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Вариант 1.
Описать класс с именем WORKER, содержащий поля:
- фамилия и инициалы работника;
- название занимаемой должности;
- зарплату;
- год поступления на работу.

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа WORKER;
– вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
– если таких работников нет, вывести соответствующее сообщение.

Вариант 2.
Описать класс с именем TRAIN, содержащий поля:
– название пункта назначения;
– номер поезда;
– время отправления.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа TRAIN;
– вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
– если таких поездов нет, вывести соответствующее сообщение.

Вариант 3.
Описать класс с именем TRAIN, содержащий поля:
– название пункта назначения;
– номер поезда;
– время отправления.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа TRAIN;
– вывод на экран информации о пункте назначения, в который отправляется поезд, номер которого введен с клавиатуры;
– если таких поездов нет, вывести соответствующее сообщение.

Вариант 4.
Описать класс с именем MARSH, содержащий поля:
– название начального пункта маршрута;
– название конечного пункта маршрута;
– номер маршрута.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа MARSH;
– вывод на экран информации о маршруте, номер которого введен с клавиатуры;
– если таких маршрутов нет, вывести соответствующее сообщение.

Вариант 5.
Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 6.
Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;

– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения приходятся на год, значение которого введено с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 7.

Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения приходятся на день, значение которого введено с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 8.

Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения совпадают с введенными с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 9.

Описать класс с именем ORDER, содержащий поля:
– расчетный счет плательщика;
– расчетный счет получателя;
– перечисляемая сумма в руб.

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа ORDER;
– вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры;
– если таких счетов нет, вывести соответствующее сообщение.

Вариант 10.

Описать класс с именем STUDENT, содержащий поля:
– фамилия и инициалы;
– номер группы;
– успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа STUDENT;
– вывод на дисплей фамилий и номеров групп для всех студентов, если средний балл студента больше 4.0;

– если таких студентов нет, вывести соответствующее сообщение.

Вариант 11.
Описать класс с именем STUDENT, содержащий поля:
– фамилия и инициалы;
– номер группы;
– успеваемость (массив из пяти элементов).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа STUDENT;
– вывод на дисплей фамилий и номеров групп для всех студентов, если они имеют оценки 4 и 5;
– если таких студентов нет, вывести соответствующее сообщение.

Вариант 12.
Описать класс с именем STUDENT, содержащий поля:
– фамилия и инициалы;
– номер группы;
– успеваемость (массив из пяти элементов).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа STUDENT;
– вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
– если таких студентов нет, вывести соответствующее сообщение.

Вариант 13.
Описать класс с именем AEROFLOT, содержащий поля:
– название пункта назначения рейса;
– номер рейса;
– тип самолета.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа AEROFLOT;
– вывод на экран информации о рейсе, номер которого введен с клавиатуры;
– если таких рейсов нет, вывести соответствующее сообщение.

Вариант 14.
Описать класс с именем AEROFLOT, содержащий поля:
– название пункта назначения рейса;
– номер рейса;
– тип самолета.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа AEROFLOT;
– вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры;
– если таких рейсов нет, вывести соответствующее сообщение.

Вариант 15.
Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).
Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 16.
Описать класс с именем WORKER, содержащий поля:
– фамилия и инициалы работника;
– название занимаемой должности;
– зарплату;
– год поступления на работу.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа WORKER;
– вывод на дисплей фамилий работников, чья зарплата выше средней по организации и стаж работы больше трех лет;
– если таких работников нет, вывести соответствующее сообщение.

Вариант 17.
Описать класс с именем TRAIN, содержащий поля:
– название пункта назначения;
– номер поезда;
– время отправления.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа TRAIN;
– вывод на экран информации о поездах, отправляющихся в пункт назначения введенного с клавиатуры;
– если таких поездов нет, вывести соответствующее сообщение.

Вариант 18.
Описать класс с именем TRAIN, содержащий поля:
– название пункта назначения;
– номер поезда;
– время отправления.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа TRAIN;
– вывод на экран информации о пункте назначения, в который отправляется поезд, номер которого введен с клавиатуры;
– если таких поездов нет, вывести соответствующее сообщение.

Вариант 19.
Описать класс с именем MARSH, содержащий поля:
– название начального пункта маршрута;
– название конечного пункта маршрута;
– номер маршрута.
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа MARSH;
– вывод на экран информации о маршруте, номер которого введен с клавиатуры;
– если таких маршрутов нет, вывести соответствующее сообщение.

Вариант 20.
Описать класс с именем NOTE, содержащий поля:

– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения приходятся на год, значение которого введено с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.