Лабораторная работа №1. Классы и объекты в языке C#

Срок выполнения – 2 недели.

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.

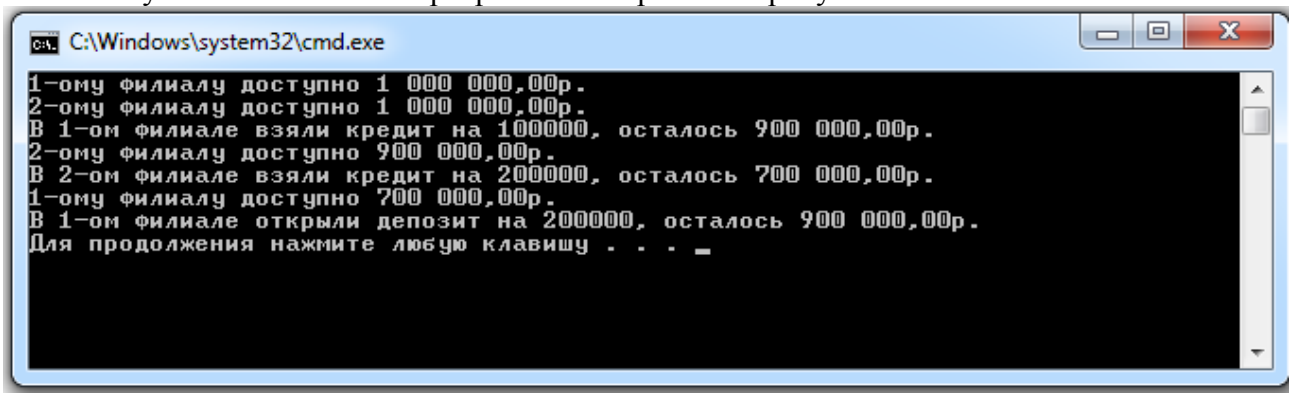

Рис. 2.2 Результат выполнения программы.

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.

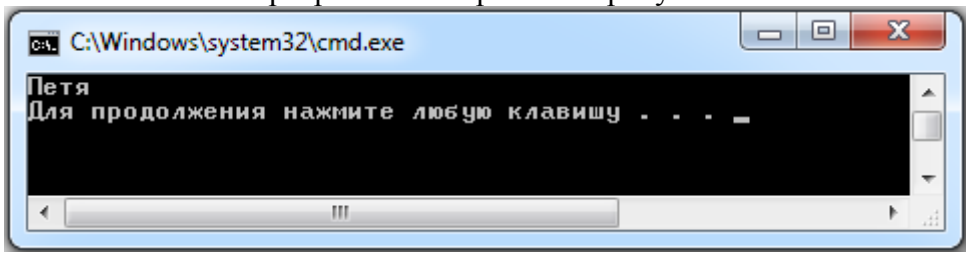

Рис. 2.3 Результат выполнения программы.

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.

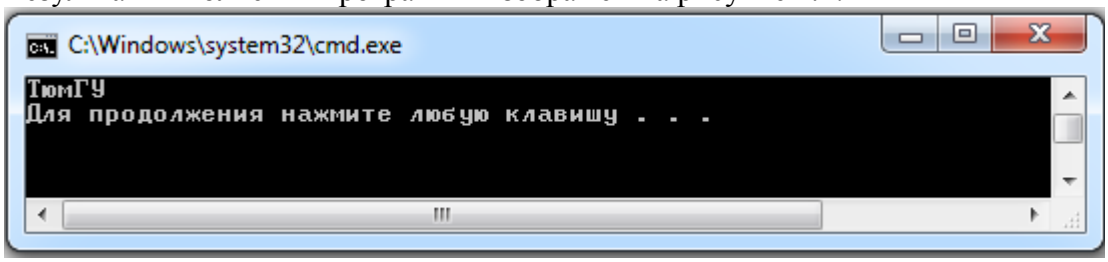

Рис. 2.4 Результат выполнения программы.

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.

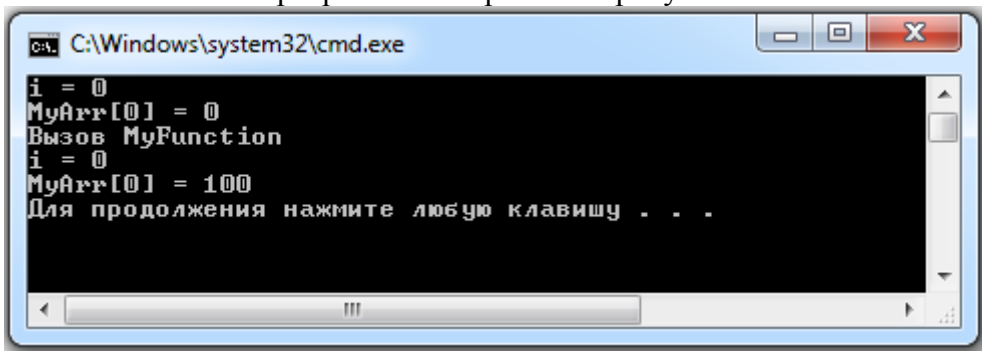

Рис. 2.5 Результат выполнения программы.

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.

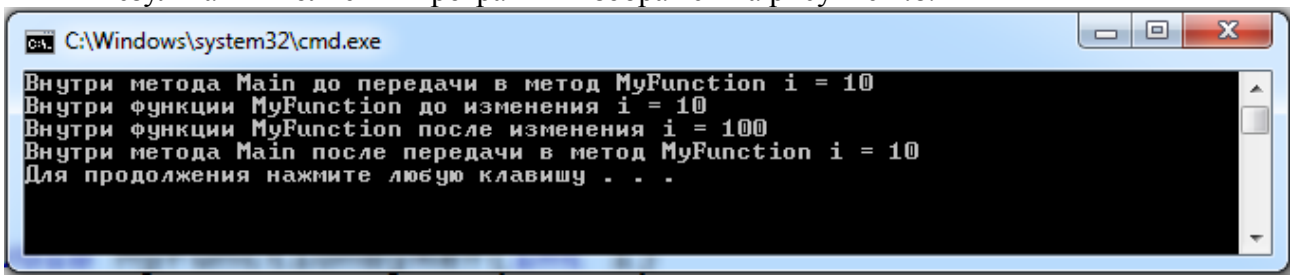

Рис. 2.6 Результат выполнения программы.

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.

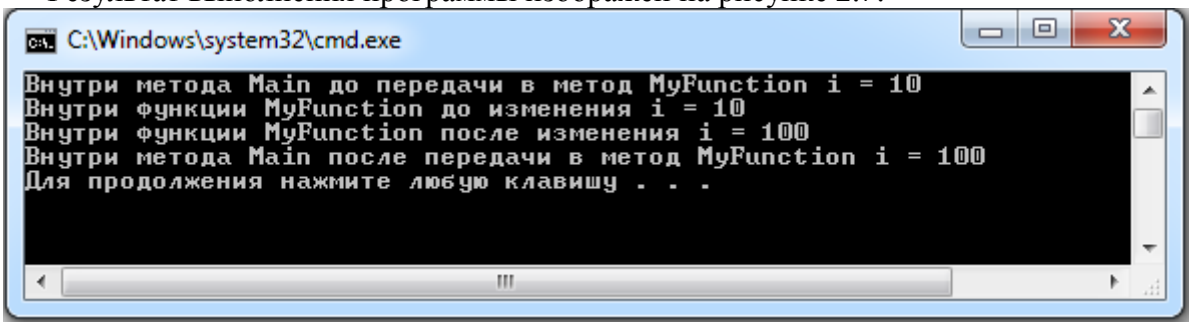

Рис. 2.7 Результат выполнения программы.

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.

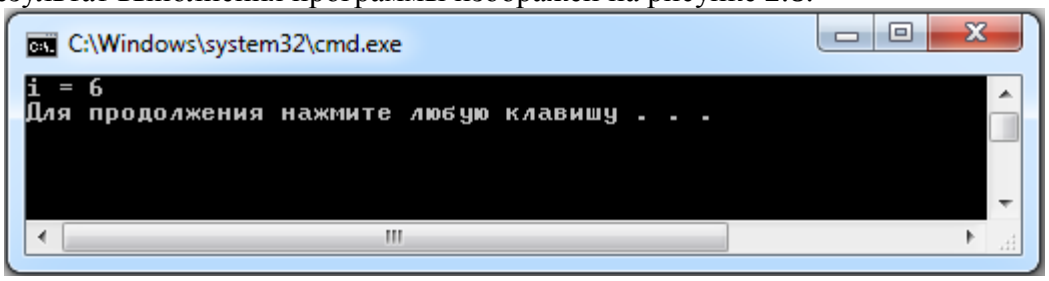

Рис. 2.8 Результат выполнения программы.

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.

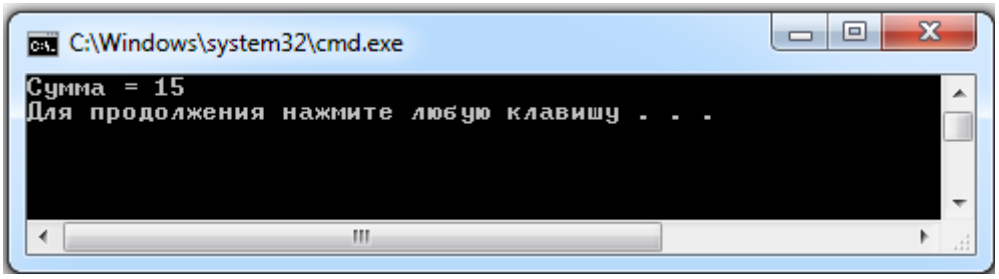

Рис. 2.9 Результат выполнения программы.

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.

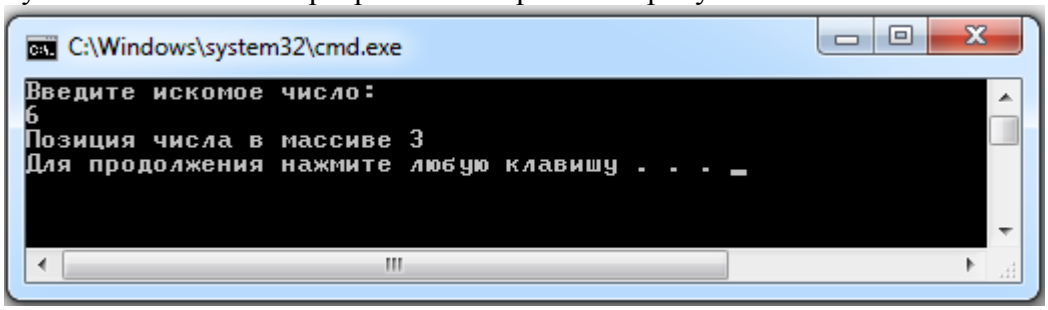

Рис. 2.10 Результат выполнения программы.

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.

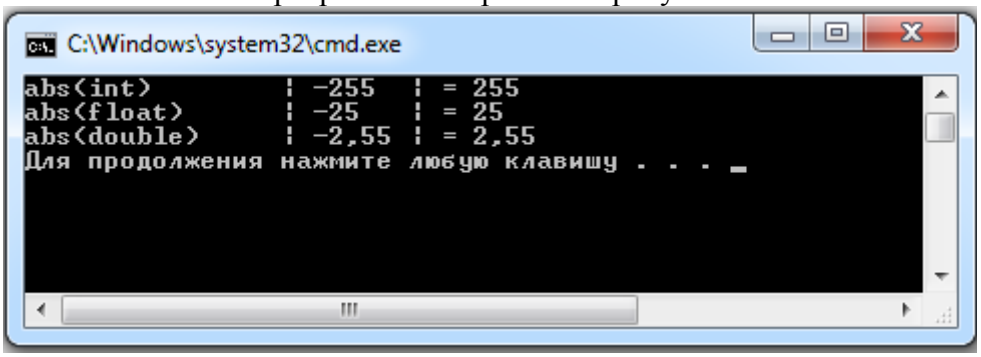

Рис. 2.11 Результат выполнения программы.

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.

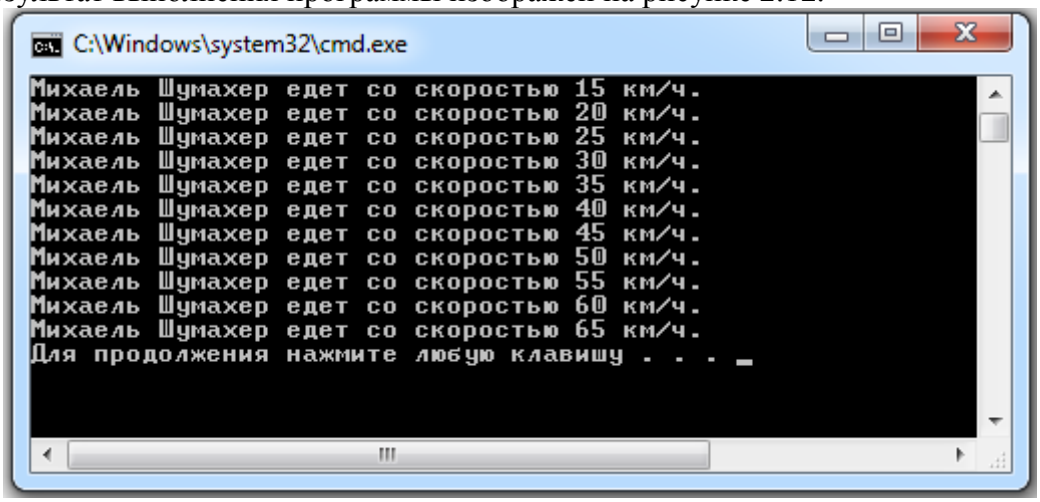

Рис. 2.12 Результат выполнения программы.

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.

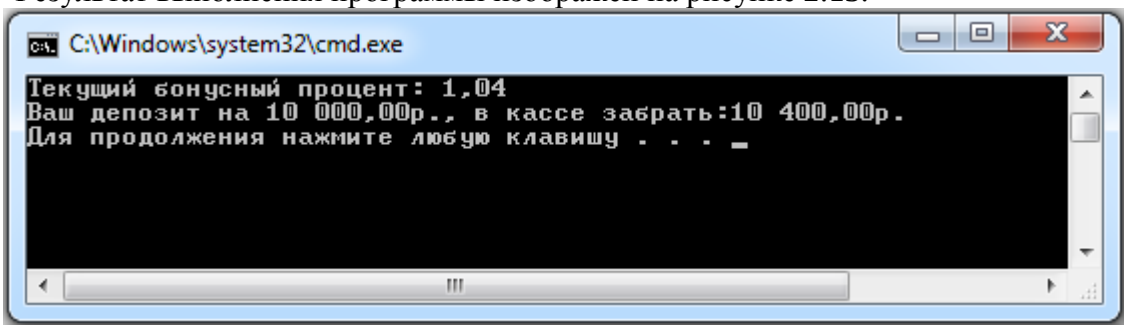

Рис. 2.13 Результат выполнения программы.

2.8 Перезагрузка операторов

Введение в перегрузку операторов

Перегрузка операторов позволяет указать, как стандартные операторы будут использоваться с объектами класса. Перегрузка

Требования к перегрузке операторов:

- перегрузка операторов должна выполняться открытыми статическими методами класса (спецификаторы public static);
- у метода - оператора тип возвращаемого значения или одного из параметров должен совпадать с типом, в котором выполняется перегрузка оператора;
- параметры метода - оператора не должны включать модификатор out и ref.

Таким образом, невозможно изменить значение стандартных операций для стандартных типов данных.

Таблица 1. Операторы, допускающие перегрузку.

Операторы	**Категория операторов**
-	Изменение знака переменной
!	Операция логического отрицания
~	Операция побитового дополнения, которая приводит к инверсии каждого бита
++, --	Инкремент и декремент
true, false	Критерий истинности объекта, определяется разработчиком класса
+, –, *, /, %	Арифметические операторы
&, |, ^, <<, >>	Битовые операции
==, !=, <, >, <=, >=	Операторы сравнения
&&, ||	Логические операторы
[]	Операции доступа к элементам массивов моделируются за счет индексаторов
()	Операции преобразования

Таблица 2. Операторы, не допускающие перегрузку.

Операторы	Категория операторов
+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Перегружаются автоматически при перегрузке соответствующих бинарных операций
=	Присвоение
.	Доступ к членам типа
?:	Оператора условия
new	Создание объекта
as, is, typeof	Используются для получения информации о типе
–>, sizeof,*,&	Доступны только в небезопасном коде

Перегрузка операторов имеет некоторые ограничения:

- перегрузка не может изменить приоритет операторов;
- при перегрузке невозможно изменить число операндов, с которыми работает оператор;
- не все операторы можно перегружать (таблицы 1 и 2).

Перегрузку операторов можно использовать как в классах, так и в структурах.

Поскольку перегруженные операторы являются статическими методами, они не получают указателя this, поэтому унарные операторы должны получать 1 параметр, бинарные 2.

Синтаксис перегрузки:

public static <тип результата>

operator <символ операции> (параметры)

Перегрузка унарных операторов

Можно определять в классе следующие унарные операции:

+ - ! ~ ++ -- true false

Синтаксис объявителя унарной операции:

тип operator унарнаяоперация (параметр)

Примеры заголовков унарных операций:

public static int operator +(MyObject m)

public static MyObject operator - - (MyObject m)

public static bool operator true(MyObject m)

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

- для операций +, -, ! и - величину любого типа;
- для операций ++ и -- величину типа класса, для которого она определяется;
- для операций true и false величину типа bool.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

Пример операторов инкремента, декремента и изменения знака -.

Класс Point описывает точку на плоскости, точка имеет координаты x и y. Оператор ++ увеличивает обе координаты на 1, оператор -- уменьшает, оператор – изменяет знак координат на противоположный.

```
namespace UnaryOperator
   {
    //класс точки на плоскости – пример для перегрузки операторорв
    class CPoint
     {
      int x, y;
```

```
  public CPoint(int x, int y)
    { this.x = x; this.y = y; }
  //перегрузка инкремента
  public static CPoint operator ++(CPoint s)
    { s.x++; s.y++; return s;}
  //перегрузка декремента
  public static CPoint operator --(CPoint s)
    { s.x--; s.y--; return s; }
  //перегрузка оператора -
  public static CPoint operator -(CPoint s)
    {
     CPoint p = new CPoint(s.x, s.y);
     p.x = -p.x; p.y = -p.y; return p;
    }
  public override string ToString()
    {
     return string.Format("X = {0} Y = {1}", x, y);
    }
 }
class Test
 {
  static void Main()
    {
     CPoint p = new CPoint(10, 10);
     //префиксная и постфиксная формы выполняются одинаково
     Console.WriteLine(++p); //x=11, y=11
     CPoint p1 = new CPoint(10, 10);
     Console.WriteLine(p1++); //x=11, y=11
     Console.WriteLine(--p); //x=10, y=10
     Console.WriteLine(-p); //x=-10, y=-10
     //после выполнения оператора –
     //состояние исходного объекта не изменилось
     Console.WriteLine(p); //x=10, y=10
    }
  }
```

Результат выполнения программы изображен на рисунке 2.1.

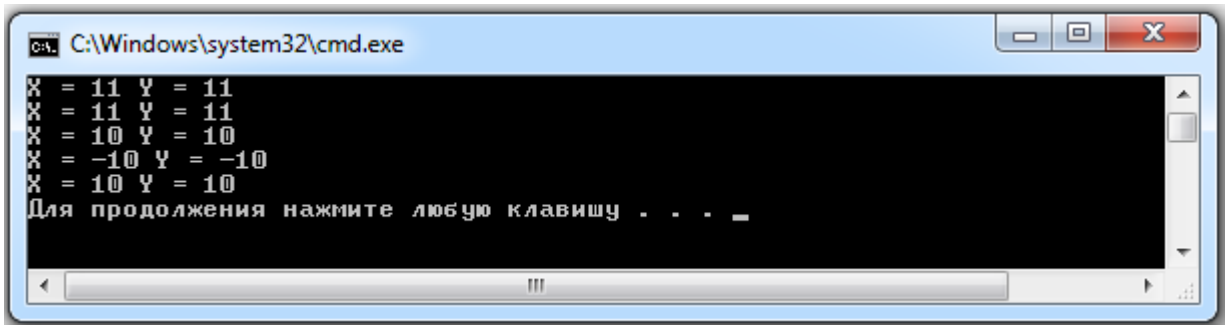

Рис. 2.1 Результат выполнения программы.

В данном примере CPoint является ссылочным типом, поэтому изменения значений x и y, которые выполняются в перегруженных операторах инкремента и декремента, изменяют переданный в них объект. Оператор – (изменение знака) не должен изменять состояние переданного объекта, а должен возвращать новый объект с измененным знаком. Для этого в реализации этого метода создается новый объект CPoint, изменяется знак его координат и этот объект возвращается из метода.

В C# нет возможности выполнить отдельно перегрузку постфиксной и префиксной форм операторов инкремента и декремента. Поэтому при вызове постфиксная и префиксная форма работают одинаково.

При перегрузке операторов true и false разработчик задает критерий истинности для своего типа данных. После этого объекты типа напрямую можно использовать в структуре операторов if, do, while, for в качестве условных выражений.

Перегрузка выполняется по следующим правилам:

- оператор true должен возвращать значение true, если состояние объекта истинно и false в противном случае;
- оператор false должен возвращать значение true, если состояние объекта ложно и false в противном случае;
- операторы true и false надо перегружать в паре.

При этом возможна ситуация, когда состояние не является не истинным ни ложным, т.е. оба оператора могут вернуть результат false.

При перегрузке операторов true и false используется следующая таблица истинности (табл. 3).

Таблица 3. Таблица истинности при перегрузке операторов true и false.

Значение	Оператор True	Оператор False
1	true	False
-1	false	True
0	false	false

```
public struct DBBool
   {
      //три возможных значения
      // Значение параметра может быть: - 1(false), 1 - true и 0 - null.
      public static readonly DBBool Null = new DBBool(0);
      public static readonly DBBool False = new DBBool(-1);
      public static readonly DBBool True = new DBBool(1);
      sbyte value;
      DBBool(int value)
      {  this.value = (sbyte)value; }
      public DBBool(DBBool b)
      { this.value = (sbyte)b.value; }
      // Возвращает true, если в операнде содержится True,
      // иначе возвращает false
      public static bool operator true(DBBool x)
      { return x.value > 0; }
      // Возвращает true, если в операнде содержится False,
      // иначе возвращает false
      public static bool operator false(DBBool x)
      { return x.value < 0; }
   }
   class Test
   {
      static void Main()
      {
         DBBool b1 = new DBBool(DBBool.True);
         if (b1) Console.WriteLine("b1 is true");
         else Console.WriteLine("b1 is not true");
```

```
    }
  }
```

Как видно из реализации, если в объекте класса содержится значение null, то оба оператора возвращают значение false.

Перегрузка бинарных операторов

Можно определять в классе следующие бинарные операции:

+ - * / % & ≪ ≫ == != > < >= <=

Синтаксис объявителя бинарной операции:

тип operator бинарная_операция (параметр1, параметр2)

Примеры заголовков бинарных операций:

Cjblic static MyObject operator + (MyObject ml, MyObject m2)

Cublic static bool operator == (MyObject ml, MyObject m2)

Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа.

Операции == и !=, > и <, >= и <= определяются только парами и обычно возвращают логическое значение. Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок, как определено по умолчанию для ссылочных типов.

Пример перегрузки бинарных операций:

```
namespace BinaryOperator
 {
  class CVector
  {
    public int x;
    public int y;
    public CVector(int x, int y)
     { this.x = x; this.y = y; }
    public override string ToString()
     { return string.Format("Vector: X = {0} Y = {1}", x, y); }
  }
 class CPoint
  {
    private int x;
    private int y;
```

```
    public CPoint(int x, int y)
     { this.x = x; this.y = y; }
    //перегрузка бинарного оператора +
    public static CPoint operator +(CPoint p, CVector v)
     { return new CPoint(p.x + v.x, p.y + v.y); }
    //перегрузка бинарного оператора *
    public static CPoint operator *(CPoint p, int a)
     { return new CPoint(p.x * a, p.y * a); }
    //перегрузка бинарного оператора -
    public static CVector operator -(CPoint p1, CPoint p2)
     { return new CVector(p1.x - p2.x, p1.y - p2.y); }
    public override string ToString()
     { return string.Format("Point: X = {0} Y = {1}", x, y); }
  }
  class Program
  {
   static void Main(string[] args)
   {
      CPoint p1 = new CPoint(10, 10);
      CPoint p2 = new CPoint(12, 20);
      CVector v = new CVector(10, 20);
      Console.WriteLine("Точка p1: {0}", p1);
      Console.WriteLine("Сдвиг: {0}", p1 + v);
      Console.WriteLine("Масштабирование: {0}", p1 * 10);
      Console.WriteLine("Точка p2: {0}", p2);
      Console.WriteLine("Расстояние: {0}", p2 - p1);
   }
  }
 }
```

Результат выполнения программы изображен на рисунке 2.2.

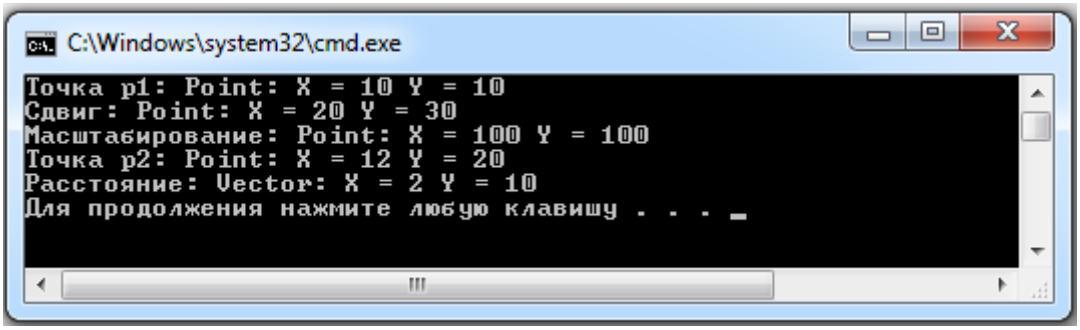

Рис. 2.2 Результат выполнения программы.

Выполненная перегрузка автоматически перегружает операторы +=, *=, -=. Например, можно записать: p1+= v;

Для реализации этого действия будет использован перегруженный оператор +.

Однако перегруженные в примере операторы будут использоваться компилятором только если переменная типа CPoint находится слева от знака операнда. Т.е. выражение p * 10 откомпилируется нормально, а при перестановке сомножителей, т.е. в выражении 10 * p произойдет ошибка компиляции. Для исправления этой ошибки следует перегрузить оператор * с другим порядком операндов:

```
public static CPoint operator *(int a, CPoint p)
{ return p * a; }
```

При перегрузке операторов отношения надо учитывать, что есть два способа проверки равенства:

- равенство ссылок (тождество);
- равенство значений.

В классе Object определены следующие методы сравнения объектов:

```
public static bool ReferenceEquals(Object obj1, Object obj2)
public bool virtual Equals(Object obj)
```

Есть отличия в работе этих методов со значимыми и ссылочными типами.

Метод ReferenceEquals() проверяет, указывают ли две ссылки на один и тот же экземпляр класса; точнее - содержат ли две ссылки один и тот адрес памяти. Этот метод невозможно переопределить. Со значимыми типами ReferenceEquals() всегда возвращает false, т.к. при сравнении выполняется приведение к Object и упаковка, упакованные объекты располагаются по разным адресам.

Метод Equals() является виртуальным. Его реализация в Object выполняется проверку равенства ссылок, т.е. работает так же как и ReferenceEquals. Для значимых типов в базовом типе System.ValueType выполнена перегрузка метода Equals(), которая выполняет сравнение объектов путем сравнения всех полей (побитовое сравнение).

Пример использования операторов ReferenceEquals() и Equals() со ссылочными и значимыми типами:

```
namespace Equals_and_ReferenceEquals
  {
    class CPoint
    {   private int x, y;
        public CPoint(int x, int y)
        { this.x = x; this.y = y; }
    }
    struct SPoint
    {
        private int x, y;
        public SPoint(int x, int y)
        { this.x = x; this.y = y; }
    }
    class Program
    {
        static void Main()
        {
 //Работа метода ReferenceEquals с ссылочным и значимым типами
            //ссылочный тип
            CPoint p = new CPoint(0, 0);
            CPoint p1 = new CPoint(0, 0);
            CPoint p2 = p1;
            Console.WriteLine("ReferenceEquals(p, p1)= {0}",
            ReferenceEquals(p, p1)); //false
            //хотя p,p1содержат одинаковые значения,
            //они указывают на разные адреса памяти
            Console.WriteLine("ReferenceEquals(p1, p2)= {0}",
            ReferenceEquals(p1, p2));//true
            //p1 и p2 указывают на один и тот же адрес памяти
            //значимый тип
            SPoint p3 = new SPoint(0, 0);
    //при передаче в метод ReferenceEquals выполняется упаковка,
    //упакованные объекты располагаются по разным адресам
```

```
            Console.WriteLine("ReferenceEquals(p3, p3) = {0}",
            ReferenceEquals(p3, p3)); //false
            //Работа метода Equals с ссылочным и значимым типами
            //ссылочный тип
            CPoint cp = new CPoint(0, 0);
            CPoint cp1 = new CPoint(0, 0);
        Console.WriteLine("Equals(cp, cp1) = {0}", Equals(cp, cp1)); //false
            //выполняется сравнение адресов значимый тип
            SPoint sp = new SPoint(0, 0);
            SPoint sp1 = new SPoint(0, 0);
        Console.WriteLine("Equals(sp, sp1) = {0}", Equals(sp, sp1)); //true
            //выполняется сравнение значений полей
          }
        }
    }
```

Результат выполнения программы изображен на рисунке 2.3.

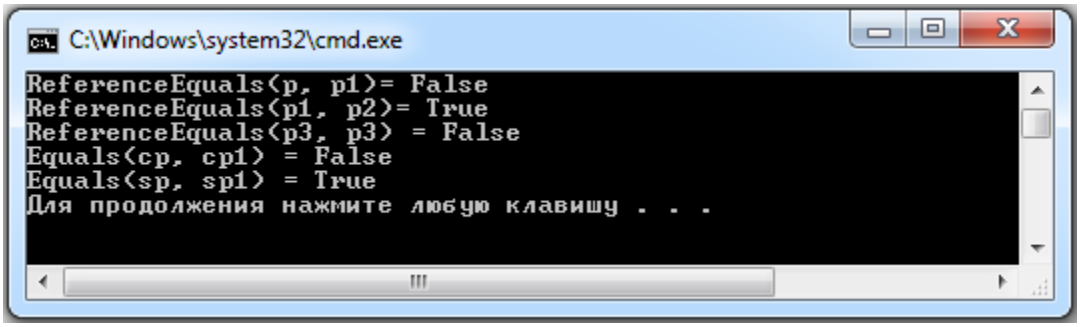

Рис. 2.3 Результат выполнения программы.

При создании собственного типа оператор Equals() можно перегрузить. Для ссылочных типов перегрузку следует выполнять, только если тип представляет собой неизменяемый объект. Например, для типа String, который содержит в себе неизменную строку, имеется перегруженный метод Equals() и оператор ==.

Поскольку в System.ValueType перегруженный метод Equals() выполняет побитовое сравнение, то в собственных значимых типах его можно не перегружать. Однако, в System.ValueType получение значений полей для сравнения в методе Equals() выполняется с помощью рефлексии, что приводит к снижению производительности. Поэтому при разработке значимого типа для увеличения быстродействия рекомендуется выполнить перегрузку метода Equals().

При перегрузке метода Equals() следует также перегружать метод GetHashCode(). Этот метод предназначен для получения целочисленного значения хеш - кода объекта. Причем,

различным (т.е. не равным между собой) объектам должны соответствовать различные хеш – коды. Если перегрузку метода GetHashCode() не выполнить возникнет предупреждение компилятора.

Перегрузка оператора == обычно выполняется путем вызова метода Equals().

Если предполагается сравнивать экземпляры собственного типа для целей сортировки, рекомендуется унаследовать этот тип от интерфейсов System.IComparable и System.IComparable<T> и реализовать метод CompareTo(). В дальнейшем этот метод можно вызывать из реализации Equals() и возвращать true, если CompareTo() возвращает 0.

Пример перегрузки операторов == и != для класса CPoint:

```
namespace ComparisonOperator
  {
    using System;
    class CPoint
    {
      int x, y;
      public CPoint(int x, int y)
      { this.x = x; this.y = y; }
      //Перегрузка метода Equals
      public override bool Equals(object obj)
      {
        //если obj == null,
  //значит он != объекту, от имени которого вызывается этот метод
        if (obj == null) return false;
        CPoint p = obj as CPoint;
        //переданный объект не является ссылкой на CPoint
        if (p == null) return false;
        //проверяется равенство содержимого
        return ((x == p.x) && (y == p.y));
      }
    //При перегрузке Equals надо также перегрузить GetHashCode()
      public override int GetHashCode()
      {
        return x ^ y;//использование XOR для получения хеш кода
      }
      public static bool operator ==(CPoint p1, CPoint p2)
```

```
        {
      //проверка, что переменные ссылаются на один и тот же адрес
            //сравнение p1 == p2 приведет к бесконечной рекурсии
            if (ReferenceEquals(p1, p2)) return true;
            //приведение к object необходимо,
     //т.к. сравнение p1 == null приведет к бесконечной рекурсии
            if ((object)p1 == null) return false;
            return p1.Equals(p2);
        }
        public static bool operator !=(CPoint p1, CPoint p2)
        { return (!(p1 == p2)); }
    }
    class Program
    {
        static void Main(string[] args)
        {
            CPoint cp = new CPoint(0, 0);
            CPoint cp1 = new CPoint(0, 0);
            CPoint cp2 = new CPoint(1, 1);
            Console.WriteLine("cp == cp1: {0}", cp == cp1); //true
            Console.WriteLine("cp == cp1: {0}", cp == cp2); //false
        }
    }
}
```

Условные логические операторы && и || нельзя перегрузить, но они вычисляются с помощью & и |, допускающих перегрузку.

Пример класса DBBool использующего перегрузку логических операторов:

```
public struct DBBool
   {
      //три возможных значения
      // Значение параметра может быть: - 1(false), 1 - true и 0 - null.
      public static readonly DBBool Null = new DBBool(0);
      public static readonly DBBool False = new DBBool(-1);
      public static readonly DBBool True = new DBBool(1);
      sbyte value;
```

```
DBBool(int value)
{ this.value = (sbyte)value; }
public DBBool(DBBool b)
{ this.value = (sbyte)b.value; }
// Возвращает true, если в операнде содержится True,
// иначе возвращает false
public static bool operator true(DBBool x)
{ return x.value > 0; }
// Возвращает true, если в операнде содержится False,
// иначе возвращает false
public static bool operator false(DBBool x)
{ return x.value < 0; }
// Оператор Логическое И. Возвращает:
// False, если один из операндов False независимо
// от 2-ого операнда
// Null, если один из операндов Null, а другой Null или True
// True, если оба операнда True
public static DBBool operator &(DBBool x, DBBool y)
{ return new DBBool(x.value < y.value ? x.value : y.value); }
// Оператор Логическое ИЛИ. Возвращает:
// True, если один из операндов True независимо
// от 2-ого операнда
// Null, если один из операндов Null, а другой False или Null
// False, если оба операнда False
public static DBBool operator |(DBBool x, DBBool y)
{ return new DBBool(x.value > y.value ? x.value : y.value); }
// Оператор Логической инверсии. Возвращает:
// False, если операнд содержит True
// True, если операнд содержит False
// Null, если операнд содержит Null
public static DBBool operator !(DBBool x)
{ return new DBBool(-x.value); }
public override string ToString()
{
    if (value > 0) return "DBBool.True";
```

```
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
class Test
{
    static void Main()
    {
        DBBool bTrue = new DBBool(DBBool.True);
        DBBool bNull = new DBBool(DBBool.Null);
        DBBool bFalse = new DBBool(DBBool.False);
        Console.WriteLine("bTrue && bNull is {0}", bTrue && bNull);
        Console.WriteLine("bTrue && bFalse is {0}", bTrue && bFalse);
        Console.WriteLine("bTrue && bTrue is {0}", bTrue && bTrue);
        Console.WriteLine();
        Console.WriteLine("bTrue || bNull is {0}", bTrue || bNull);
        Console.WriteLine("bFalse || bFalse is {0}", bFalse || bFalse);
        Console.WriteLine("bTrue || bFalse is {0}", bTrue || bFalse);
        Console.WriteLine();
        Console.WriteLine("!bTrue is {0}", !bTrue);
        Console.WriteLine("!bFalse is {0}", !bFalse);
        Console.WriteLine("!bNull is {0}", !bNull);
    }   }
```

Результат выполнения программы изображен на рисунке 2.4.

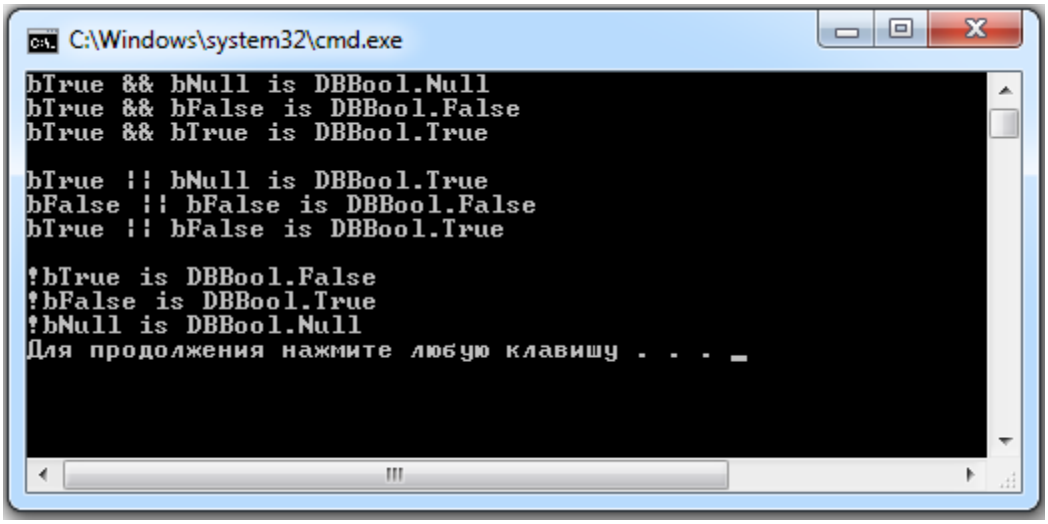

Рис. 2.4 Результат выполнения программы.

Перегрузка операторов преобразования типа

Операции преобразования типа обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных. Синтаксис объявителя операции преобразования типа:

implicit operator тип (параметр) // неявное преобразование

explicit operator тип (параметр) // явное преобразование

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразования либо типа класса к другому типу, либо наоборот. Преобразуемые типы не должны быть связаны отношениями наследования.

Конструктор с 1-им параметром не используется для преобразования произвольного типа в собственный тип. Для ссылочных и значимых типов приведение выполняется одинаково.

Приведение может выполняться явным и неявным образом. Явное приведение типов требуется, если возможна потеря данных в результате приведения. Например:

- при преобразовании int в short, потому что размер short недостаточен для сохранения значения int;
- при преобразовании типов данных со знаком в беззнаковые может быть получен неверный результат, если переменная со знаком содержит отрицательное значение;
- при конвертировании типов с плавающей точкой в целые дробная часть теряется;
- при конвертировании типа, допускающего null-значения, в тип, не допускающий null, если исходная переменная содержит null, генерируется исключение.

Если потери данных в результате приведения не происходит приведение можно выполнять как неявное.

Пример:

Приведение CPoint к типу int с потерей точности, к типу double без потери точности. В качестве результата возвращается расстояние от точки до начала координат.

Приведение типа int к CPoint без потери точности, типа double к CPoint с потерей точности. В качестве результата возвращается точка, содержащее заданное значение в качестве значений координат.

namespace CastOperator

```
{
    class CPoint
    {
        int x, y;
        public CPoint(int x, int y)
        { this.x = x; this.y = y; }
//может быть потеря точности, преобразование должно быть явным
        public static explicit operator int(CPoint p)
        {
            return (int)Math.Sqrt(p.x * p.x + p.y * p.y);
            //можно и так: return (int)(double)p;
        }
        //преобразование без потери точности, может быть неявным
        public static implicit operator double(CPoint p)
        { return Math.Sqrt(p.x * p.x + p.y * p.y); }
        //переданное значение сохраняется в x и y координате,
        //преобразование без потери точности, может быть неявным
        public static implicit operator CPoint(int a)
        { return new CPoint(a, a); }
        //преобразование с потерей точности, должно быть явным
        public static explicit operator CPoint(double a)
        {
          return new CPoint((int)a, (int)a);
         }
        public override string ToString()
        {
         return string.Format("X = {0} Y = {1}", x, y);
         }
    }
    class Test
    {
        static void Main()
        {
          CPoint p = new CPoint(2, 2);
           //выполнение явного преобразования CPoint в int
```

```
            int a = (int)p;
            //выполнение неявного преобразования CPoint в double
            double d = p;
            Console.WriteLine("p as int: {0}", a); //2
            Console.WriteLine("p as double: {0:0.0000}", d);          //2.8284
            p = 5; //выполнение неявного преобразования int в CPoint
            Console.WriteLine("p: {0}", p); //x = 5 y = 5
           //выполнение явного преобразования double в CPoint
            p = (CPoint)2.5;
            Console.WriteLine("p: {0}", p); //x = 2 y = 2
        }
    }
}
```

Результат выполнения программы изображен на рисунке 2.5.

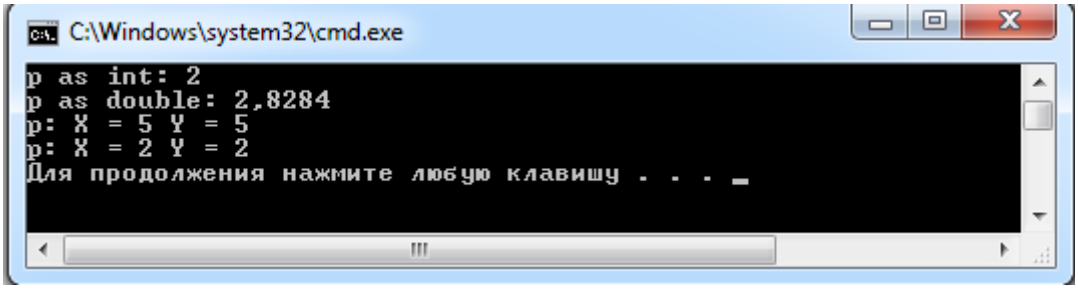

Рис. 2.5 Результат выполнения программы.

Имеется возможность выполнять приведение между экземплярами разных собственных структур или классов. Однако при этом существуют следующие ограничения:

- нельзя определить приведение между классами, если один из них является наследником другого;
- приведение может быть определено только в одном из типов: либо в исходном типе, либо в типе назначения.

Например, имеется следующая иерархия классов (рис. 2.6).

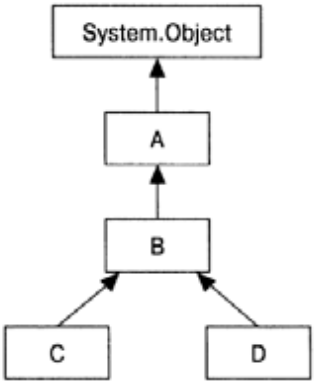

Рис. 2.6 Пример иерархии классов.

Единственное допустимое приведение типов – это приведения между классами C и D, потому что эти классы не наследуют друг друга. Код таких приведений может выглядеть следующим образом:

public static explicit operator D(C value) {…}

public static explicit operator C(D value) {…}

Эти операторы могут быть внутри определения класса C или же внутрь определения класса D. Если приведение определено внутри одного класса, то нельзя определить такое же приведение внутри другого.

2.9 Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки.

Синтаксис свойства:

[атрибуты] [спецификаторы] тип имя_свойства

{

[get код_доступа]

[set код_доступа]

}

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором public), поскольку они входят в интерфейс объекта.

Код доступа представляет собой блоки операторов, которые выполняются при получении (**get**) или установке (**set**) свойства. Может отсутствовать либо часть **get**, либо **set,** но не обе одновременно.

Если отсутствует часть **set**, свойство доступно только для чтения (read-only), если отсутствует часть **get**, свойство доступно только для записи (write-only).

В C# введена удобная возможность задавать разные уровни доступа для частей **get** и **set**. Например, во многих классах возникает потребность обеспечить неограниченный доступ для чтения и ограниченный — для записи.

Спецификаторы доступа для отдельной части должны задавать либо такой же, либо более ограниченный доступ, чем спецификатор доступа для свойства в целом. Например, если свойство описано как public, его части могут иметь любой спецификатор доступа, а если свойство имеет доступ protected internal, его части могут объявляться как internal, protected или private. Синтаксис свойства имеет вид

[атрибуты] [спецификаторы] тип имя_свойства

{

[[атрибуты] [спецификаторы] get код_доступа]

[[атрибуты] [спецификаторы] set код_доступа]

}

Пример описания свойств:

```
public class Button : Control
   {
      // закрытое поле, с которым связано свойство
      private string caption;
      public string Caption
      { // свойство
         get
         { // способ получения свойства
            return caption;
         }
         set
         { // способ установки свойства
            if (caption != value)
            {
               caption = value;
            }
         }
      }
…..
   }
```

При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод **get** должен содержать оператор **return**, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства. В методе **set** используется параметр со стандартным именем **value**, который содержит устанавливаемое значение.

Вообще говоря, свойство может и не связываться с полем. Фактически, оно описывает один или два метода, которые осуществляют некоторые действия над данными того же типа, что и свойство. В отличие от открытых полей, свойства обеспечивают разделение между

внутренним состоянием объекта и его интерфейсом и, таким образом, упрощают внесение изменений в класс.

С помощью свойств можно отложить инициализацию поля до того момента, когда оно фактически потребуется, например:

```
class A
   {
      private static ComplexObject x;// закрытое поле
      public static ComplexObject X  // свойство
      {
         get
         {
            if (x == null)
            {
              // создание объекта при 1-м обращении
               x = new ComplexObject();
            }
            return x;
         }
      }
…….
   }
```

Пример: Класс Employee реализован с четырьмя приватными полями, обозначающими имя, фамилию, возраст и зарплату сотрудника. Класс также включает два перегруженных конструктора (с параметрам и без параметров). Для каждого из полей класса предусмотрено открытое свойство с двумя методами get и set. В свойствах осуществляется дополнительная проверка задаваемых значений. Имя и фамилия приводятся к верхнему регистру, возраст проверяется на принадлежность интервалу допустимых значений, зарплат не может быть отрицательной величиной. Перегруженный метод ToString() позволяет распечатать состояние объекта.

```
class Emoloyee
   {
      private string firstName;
      private string lastName;
      private int age;
```

```
private float wage;
public Emoloyee()
{
}
public Emoloyee(string first, string last, int age, float wage)
{
   this.FirstName = first;
   this.LastName = last;
   this.Age = age;
   this.Wage = wage;
}
public string FirstName
{
   get { return firstName != null ? firstName : "Not set"; }
   set { firstName = value.ToUpper(); }
}
public string LastName
{
   get { return lastName != null ? lastName : "Not set"; ; }
   set { lastName = value.ToUpper(); }
}
public int Age
{
   get { return age; }
   set { age = (value > 100 || value < 1) ? 0 : value; }
}
public float Wage
{
   get { return wage; }
   set { wage = value < 0 ? 0 : value; }
}
public override string ToString()
{
   return string.Format("First name: {0}\nLast name:
                                    {1}\nAge:{2}\nWage: {3}\n",
```

```
        this.FirstName, this.LastName, this.Age, this.Wage);
    }
}
class Tester
{
    static void Main(string[] args)
    {
        Emoloyee emp1 = new Emoloyee("Oleg", "Sikorsky", 29,
                                                    4800F);
        Emoloyee emp2 = new Emoloyee();
        emp2.FirstName = "Daniel";
        //Last name не установлено
        //попытка присвоить невозможный возраст
        emp2.Age = 120;
        //попытка задать зарплату со знаком минус
        emp2.Wage = -1000;
        Emoloyee emp3 = new Emoloyee("Natali", "Borisova", 29,
                                                      2500F);
        Console.WriteLine(emp1.ToString());
        Console.WriteLine(emp2.ToString());
        Console.WriteLine(emp3.ToString());
    }   }
```

Результат выполнения программы изображен на рисунке 2.7.

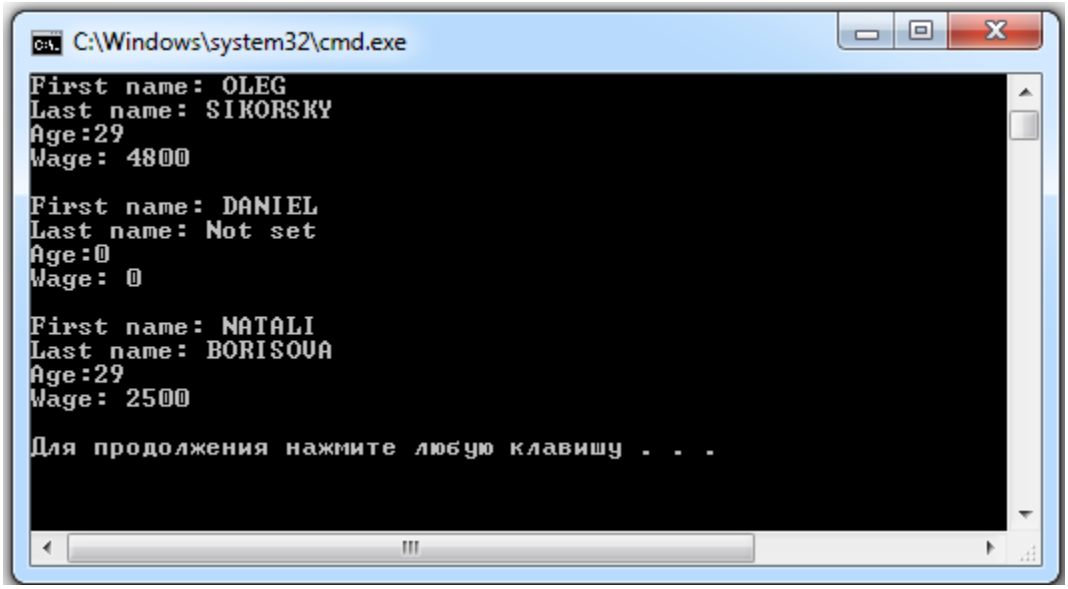

Рис. 2.7 Результат выполнения программы.

2.10 Индексаторы

Понятие индексатора

Индексатор представляет собой разновидность свойства. Если у класса есть скрытое поле, представляющее собой массив, то с помощью индексатора можно обратиться к элементу этого массива, используя имя объекта и номер элемента массива в квадратных скобках.

Объявление индексатора подобно свойству, но с той разницей, что индексаторы безымянные (вместо имени используется ссылка this) и что индексаторы включают параметры индексирования.

Синтаксис объявления индексатора следующий:

тип this [тип аргумента] {get; set;}

Тип – это тип объектов коллекции. This - это ссылка на объект, в котором появляется индексатор. Тип аргумента представляет индекс объекта в коллекции, причём этот индекс необязательно целочисленный, как мы привыкли, он может быть любого типа.

У каждого индексатора должен быть минимум один параметр, но их может быть и больше (напоминает многомерные массивы).

То, что для индексаторов используется синтаксис со ссылкой this, подчёркивает, что нельзя использовать их иначе, как на экземплярном уровне.

Создание одномерного индексатора

Пример создания и применения индексатора. Предположим, есть некий магазин, занимающийся реализацией ноутбуков. Эта ситуация отображается при помощи двух классов: класса Shop, изображающего магазин, и класса Laptop, изображающего его продукцию. Дабы не перегружать пример лишней информацией, снабдим класс Laptop только двумя полями: vendor – для отображения имени фирмы-производителя, а также price – для отображения цены ноутбука. Класс будет включать соответствующие открытые свойства Vendor и Price, конструктор с двумя параметрами, а также переопределённый метод ToString() для отображения информации по конкретной единице товара. В качестве единственного поля класса Shop выступает ссылка на массив объектов Laptop. В конструкторе с одним параметром задаётся количество элементов массива и выделяется память для их хранения. Далее нам нужно сделать возможным обращение к элементам этого массива через экземпляр класса Shop, пользуясь синтаксисом массива так, словно класс Shop и есть массив элементов типа Laptop. Для этого добавлен в класс Shop индексатор:

public Laptop this[int pos]

```
{
    get
    {
        if (pos >= LaptopArr.Length || pos < 0)
            throw new IndexOutOfRangeException();
        else
            return (Laptop)LaptopArr[pos];
    }
    set
    { LaptopArr[pos] = (Laptop)value; }
}
```

Здесь в аксессоре get мы предусматриваем выход за пределы массива, и при этом генерируется исключение IndexOutOfRangeException.

Для проверки работы классов создаётся отдельный класс Tester, содержащий точку входа. В нём создаётся экземпляр класса Shop, причём в конструкторе задается количество элементов, которые в нём можно разместить.

```
Shop sh = new Shop(3);
```

Далее мы заполняем этот массив объектами Laptop.

```
sh[0] = new Laptop("Samsung", 5200);
sh[1] = new Laptop("Asus", 4700);
sh[2] = new Laptop("LG", 4300);
```

И, наконец, вывод на экран данных по каждому объекту Laptop, пользуясь синтаксисом массива.

```
for (int i = 0; i < 3; i++)
Console.WriteLine(sh[i].ToString());
```

В цикле ограничивающим значением в условии является явно заданное число 3. Дело в том, что индексатор позволяет нам пользоваться лишь синтаксисом индексирования массива, но других функциональных возможностей массива не предоставляет. Если это был бы стандартный массив, то это условие описывается иначе:

```
for (int i = 0; i < sh.Length; i++)
. . .
```

Для того, чтобы подобная функциональная возможность появилась и в примере, необходимо добавить в класс Shop дополнительное свойство Length.

```
public int Length
{
```

```
get { return LaptopArr.Length; }
}
```

Общий вид программы:

```
namespace NS
  {
    public class Laptop
    {
      private string vendor;
      private double price;
      public string Vendor
      {
        get { return vendor; }
        set { vendor = value; }
      }
      public double Price
      {
        get { return price; }
        set { price = value; }
      }
      public Laptop(string v, double p)
      {
        vendor = v;
        price = p;
      }
      public override string ToString()
      {
        return vendor + " " + price.ToString();
      }
    }
    public class Shop
    {
      private Laptop[] LaptopArr;
      public Shop(int size)
      {
        LaptopArr = new Laptop[size];
```

```
    }
    public int Length
    {
        get { return LaptopArr.Length; }
    }
    public Laptop this[int pos]
    {
        get
        {
            if (pos >= LaptopArr.Length || pos < 0)
                throw new IndexOutOfRangeException();
            else
                return (Laptop)LaptopArr[pos];
        }
        set
        {
            LaptopArr[pos] = (Laptop)value;
        }
    }
}
public class Tester
{
    public static void Main()
    {
        Shop sh = new Shop(3);
        sh[0] = new Laptop("Samsung", 5200);
        sh[1] = new Laptop("Asus", 4700);
        sh[2] = new Laptop("LG", 4300);
        try
        {
            for (int i = 0; i < sh.Length; i++)
                Console.WriteLine(sh[i].ToString());
            Console.WriteLine();
        }
        catch (System.NullReferenceException)
```

```
        { }
      }
    }
  }
```

Результат выполнения программы изображен на рисунке 2.8.

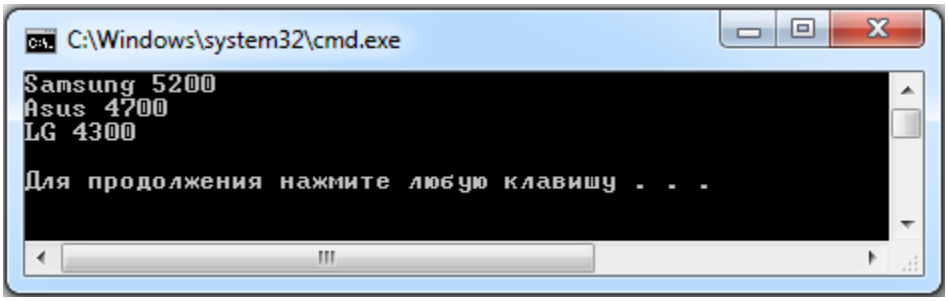

Рис. 2.8 Результат выполнения программы.

Создание многомерных индексаторов

В C# есть возможность создавать не только одномерные, но и многомерные индексаторы. Это возможно, если класс-контейнер содержит в качестве поля массив с более чем одним измерением.

Пример использования многомерного индексатора:

```
namespace NS
{
  public class A
  {
    private int[,] arr;
    private int rows, cols;
    public int Rows
    {
      get { return rows; }
    }
    public int Cols
    {
      get { return cols; }
    }
    public A(int rows, int cols)
    {
      this.rows = rows;
```

```
        this.cols = cols;
        arr = new int[rows, cols];
    }
    public int this[int r, int c]
    {
        get { return arr[r, c]; }
        set { arr[r, c] = value; }
    }
}
public class Tester
{
    static void Main()
    {
        A obj = new A(2, 3);
        for (int i = 0; i < obj.Rows; i++)
        {
            for (int j = 0; j < obj.Cols; j++)
            {
                obj[i, j] = i + j;
                Console.Write(obj[i, j].ToString());
            }
            Console.WriteLine();
        }
    }
}   }
```

2.11 Деструкторы

В C# существует специальный вид метода, называемый деструктором. Он вызывается сборщиком мусора непосредственно перед удалением объекта из памяти. В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта, например, проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т.п.).

Синтаксис деструктора:

[атрибуты] [extern] ~имя_класса()

тело

Как видно из определения, деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой (~), символизирующей обратные по отношению к конструктору действия. Тело деструктора представляет собой блок или просто точку с запятой, если деструктор определен как внешний (extern).

Сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени. Поскольку деструктор вызывается сборщиком мусора, невозможно гарантировать, что деструктор будет обязательно вызван в процессе работы программы. Следовательно, его лучше использовать только для гарантии освобождения ресурсов, а «штатное» освобождение выполнять в другом месте программы.

Применение деструкторов замедляет процесс сборки мусора.

2.12 Вложенные типы

В классе можно определять типы данных, внутренние по отношению к классу. Так определяются вспомогательные типы, которые используются только содержащим их классом. Механизм вложенных типов позволяет скрыть ненужные детали и более полно реализовать принцип инкапсуляции. Непосредственный доступ извне к такому классу невозможен (имеется в виду доступ по имени без уточнения). Для вложенных типов можно использовать те же спецификаторы, что и для полей класса.

Например, класс Monster содержит вспомогательный класс Gun. Объекты этого класса без «хозяина» бесполезны, поэтому его можно определить как внутренний:

```
using System;
namespace ConsoleApplicationi
{ class Monster
   {
     class Gun
       { ….. }
…..
   }
}
```

Помимо классов вложенными могут быть и другие типы данных: интерфейсы, структуры и перечисления.

2.13 Исключения

Иерархия исключений

Возникновение ошибок при выполнении приложения - не всегда следствие ошибок в коде приложения. Ошибки могут быть вызваны неправильными действиями пользователя или внешними причинами такими как аппаратные сбои, недоступность некоторых ресурсов (например, сетевого диска или сервера базы данных).

Таким образом, профессионально разработанное приложение должно быть готово к возникновению ошибок и должно обеспечивать их обработку.

Для обработки ошибок можно использовать различные подходы. В WinAPI многие функции при неуспешном выполнении возвращают признак ошибки (например, FALSE, INVALID_HANDLE_VALUE, NULL). Далее с помощью функции GetLastError() можно получить код ошибки и найти по этому коду ее описание. Такой способ обработки ошибок является трудоемким, т.к. после вызова функции надо проверять результат возврата, и не вполне надежным, т.к. можно продолжить работу без проверки результата, так как будто функция завершилась успешно.

В .Net Framework способ обработки ошибок значительно улучшен. Все методы при неуспешном выполнении генерируют исключения. Этот подход имеет следующие преимущества:

- Обработку ошибок можно отделить от основной логики работы программы: всю обработку ошибок можно сосредоточить в одном блоке, а не выполнять проверку после каждого вызова метода.

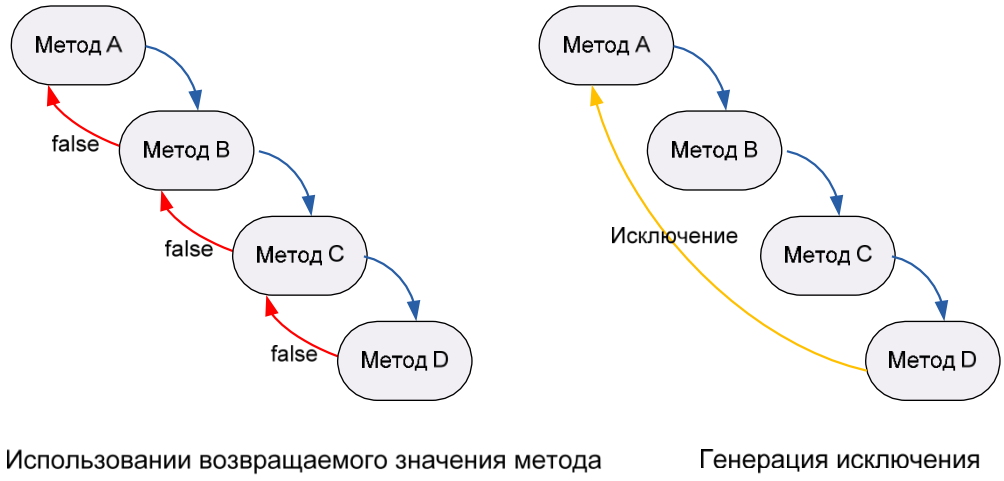

Рис. 2.9. Передача информации о возникновении ошибки по цепочке вызовов

- Возможна ситуация, когда имеется цепочка вызовов (рис. 2.9.), и в последнем методе цепочки возникает ошибка. Без обработки исключений каждый из

методов должен вернуть вызывающему методу код ошибки, в случае использования исключений при возникновении ошибки управление будет сразу передано методу, содержащему обработчик исключения.

- Исключение нельзя проигнорировать, т.к. необработанное исключение приведет к аварийному завершению программы.

Базовый класс System.Exception.

Таблица 4. Свойства класса System.Exception

Название свойства	Описание
string Message	Содержит текст сообщения с указанием причины возникновения исключения.
IDictionary Data	Ссылка на набор пар «параметр-значение». Обычно код, генерирующий исключение, добавляет записи в этот набор. Код, перехвативший исключение, может использовать эти данные для получения дополнительной информации о причине возникновения исключения.
string Source	Содержит имя сборки, сгенерировавшей исключение.
string StackTrace	Содержит имена и сигнатуры методов, вызов которых привел к возникновению исключения.
MethodBase TargetSite	Содержит метод, сгенерировавший исключение.
string HelpLink	Содержит URL документа с описанием исключения.
Exception InnerException	Указывает предыдущее исключение, если текущее было сгенерировано при обработке предыдущего исключения.

В C# исключение является объектом, который создается и «выбрасывается» (throw) в случае возникновения ошибки. CLR позволяет генерировать исключения любого типа, например Int32, String и др. CLS-совместимый язык должен быть способен генерировать и перехватывать типы исключений, производные от базового класса SystemException (таб. 4). Эти исключения называются CLS-совместимыми. Такое исключение несет дополнительную информацию об ошибке, которая облегчает отладку программы.

Существует множество классов исключений (наследников System.Exception), разработчик может использовать эти классы и также создавать собственные классы исключений. Все исключения делятся на 2 группы: SystemException и ApplicationException.

SystemException – это класс исключений, которые обычно генерируются CLR или являются исключениями общей природы и могут быть сгенерированы любым приложением.

Например, исключение StackOverflowException генерируется CLR при переполнении стека, исключение ArgumentException (и производные от него) могут быть сгенерированы любым приложением, если метод получает недопустимые значения аргументов.

ApplicationExcption – от этого класса должны наследоваться пользовательские исключения, специфичные для приложения.

Иерархия классов исключений является в некоторой степени необычной, т.к. производные классы в основном не добавляют новую функциональность к возможностям базового класса. Эти классы используются для указания более специфических причин возникновения ошибки. Например, от класса ArgumentException наследуются классы ArgumentNullException (генерируется при передаче null в качестве параметра метода) и ArgumentOutOfRange (генерируется при выходе переменной за допустимый диапазон значений).

Обработка исключений

Синтаксис блока обработки исключений:

```
try
  {
    // код, в котором может возникнуть исключение
  }
catch(тип исключения)
  {
    // обработка исключения
  }
finally
  {
     // освобождение ресурсов
  }
```

Блок try содержит код, требующий общей очистки ресурсов или восстановления после исключения.

Блок catch содержит код, который должен выполняться при возникновении исключения. При объявлении блока catch указывается тип исключения, для обработки которого он предназначен. Если блок try завершился без генерации исключения, блоки catch не выполняются. Если в блоке try не предполагается возникновение исключения, блок catch может отсутствовать, но тогда обязательно должен быть блок finally.

Блок finally обычно содержит очистку ресурсов, а также другие действия, которые необходимо гарантировано выполнить после завершения блока try и catch. Например, в этом блоке можно выполнить закрытие файла или закрытие соединения с БД. Блок finally выполняется всегда, независимо от возникновения исключения в блоке try. Если нет необходимости выполнять очистку ресурсов, блок finally может отсутствовать, но тогда обязательно должен быть блок catch. Блок try сам по себе не имеет смысла.

Для генерации исключения используется ключевое слово throw. Синтаксис генерации исключения:

throw new Конструктор класса исключения()

В качестве типа исключения надо использовать производный класс иерархии исключений, который наиболее полно описывает возникшую проблему. Не рекомендуется использовать базовые классы, т.к. в этом случае при обработке исключения трудно будет точно определить причину его возникновения.

При возникновении исключения в блоке try выполнение этого блока прекращается и CLR выполняет поиск блока catch, предназначенного для обработки исключений такого типа. Если этот блок найден, он выполняется, затем выполняется блок finally (если он есть). Если подходящий блок catch не найден, исключение считается необработанным и приводит к аварийному завершению программы.

Пример. Последовательность выполнения кода при возникновении исключения.

```
namespace ExceptionExample1
  {
    class Program
    {
      static void Main(string[] args)
      {
        try
        {
          Console.WriteLine("Before Exception"); //1
          throw new Exception("Test Exception"); //2
    Console.WriteLine("After Exception.This line will never appear");
        }
        catch (Exception e)
        { Console.WriteLine("Exception: {0}", e.Message); //3 }
        finally
        { Console.WriteLine("In finally block"); //4 }
```

```
        }
    }
}
```

Результат выполнения программы изображен на рисунке 2.10.

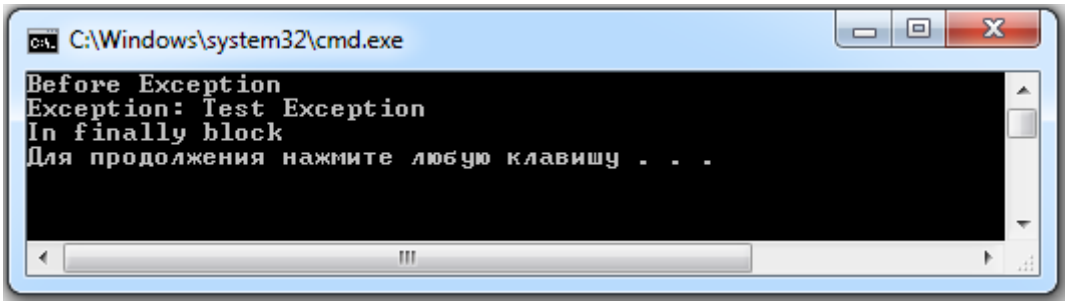

Рис. 2.10 Результат выполнения программы.

Текст «After Exception» не будет выведен, т.к. после генерации исключения выполнение блока try прекращается.

У одного блока try может быть несколько блоков catch для обработки исключений различных типов. При возникновении исключения поиск обработчика начинается с 1-ого блока catch, поэтому следует, сначала размещать обработчики для исключений производных классов, затем базовых.

Для обработки всех типов CLS совместимых исключений можно использовать блок catch, который перехватывает System.Exception, т.к. этот класс является базовым для всех классов исключений:

catch(Exception e)

{ … }

Пример. Использование нескольких блоков catch: вычисляется выражение d = 100/ln(n), n вводится пользователем. Перехватываются следующие типы исключений:

- FormatException – возникает, если введенную пользователем строку невозможно преобразовать в число.
- DivisionByZeroException – возникает, если Ln(n) = 0, т.е. n = 1
- Exception – все остальные исключения, наследуемые от Exception, например Overfloaw.

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

```
    do
    {
       try
       {
    Console.Write("{0}Input int number: ", Environment.NewLine);
          //чтение ввода пользователя
          string s = Console.ReadLine();
          //условие выхода из цикла
          if (s == string.Empty) return;
          //преобразование строки в число
          int n = Convert.ToInt32(s);
          //проверка, что полученное число принадлежит
          //области определения функции ln()
  if (n <= 0) throw new ArgumentOutOfRangeException("n <= 0");
          double f = Math.Log(n);
          int d = 100 / (int)f;
          Console.WriteLine("d = {0} f = {1}", d, f);
       }
       catch (FormatException)
       {
        //происходит, если введенное пользователем значение
        //невозможно преобразовать в целое число
          Console.WriteLine("FormatException");
       }
       catch (DivideByZeroException)
       {
          //происходит, елси Log(n) = 0 (т.е. n = 1)
          Console.WriteLine("DivideByZeroException");
       }
       catch (Exception e)
       {
 //прехват всех остальных исключений
//например, исключения ArgumentOutOfRangeException,
//которое генерируется, если Log(n) не определен (т.е. n <= 0)
          Console.WriteLine("Exception: {0}", e.Message);
```

```
            }
        }
        while (true);
    }
  }
}
```

Результат выполнения программы изображен на рисунке 2.11.

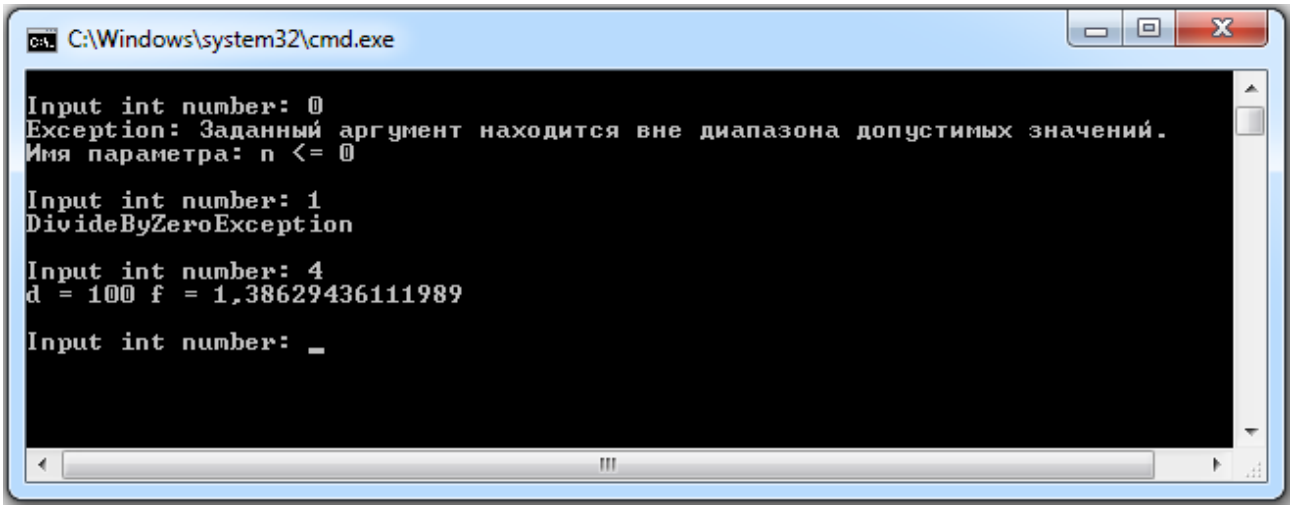

Рис. 2.11 Результат выполнения программы.

Блоки try могут быть вложенными.

Если исключение возникает во вложенном блоке, выполняется блок finally вложенного блока try, затем выполняется поиск подходящего обработчика исключения. Если исключение может быть обработано внутренним блоков catch, оно перехватывается и обрабатывается, после чего продолжается выполнение кода внешнего блока. Если не возникало нового исключения, блок catch внешнего блока игнорируется, блок finally внешнего блока выполняется в любом случае. Если исключение не может быть перехвачено внутренним блоком catch, выполняется проверка внешнего блока catch. Если этот блок не в состоянии обработать исключение, поиск подходящего обработчика выполняется выше по стеку вызовов.

Пример. Вложенные блоки try. Во внутреннем блоке происходит 2 вида исключений:

- деление на 0;
- обращение к массиву по недопустимому индексу.

1-ое исключение перехватывается внутренним блоком catch, 2-ое – внешним.

```
namespace NestedTryBlocks
  {
    class Program
    {
```

```
        static void Main()
        {
            int[] a = new int[5];
            int cnt = 0;
            try //внешний блок try
            {
                for (int i = -3; i <= 3; i++)
                {
                    //при делении на 0 не происходит выход из цикла:
                    //это исключение перехватывается
                    //и обрабатывается вложенным блоком try
                    try //вложенный блок try
                    {
                        a[cnt] = 100 / i;
                        Console.WriteLine(a[cnt]);
                        cnt++;
                    }
                    catch (DivideByZeroException e)
                    {
                        Console.WriteLine("In inner catch");
                        Console.WriteLine(e.Message);
                    }
                }
            }
            catch (IndexOutOfRangeException e)
            {
                Console.WriteLine("In outer catch");
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Результат выполнения программы изображен на рисунке 2.12.

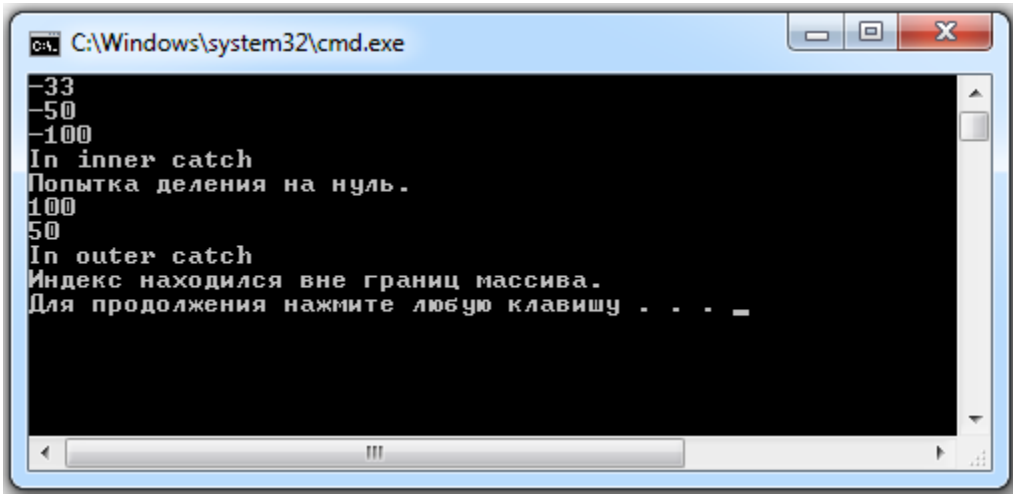

Рис. 2.12 Результат выполнения программы.

Применение конструкций checked и unchecked

При выполнении арифметических операций с целочисленными типами данных может возникать переполнение, т.е. ситуация при которой количество двоичных разрядов полученного результата превышает разрядность переменной, в которую этот результат записывается.

Переполнение может возникнуть в следующих ситуациях:

- в выражениях, которые используют арифметические операторы ++ -- - (унарный) + - * /;
- при выполнении явного преобразования целочисленных типов.

При возникновении переполнения CLR может использовать один из двух вариантов:

- проигнорировать переполнение и отбросить старшие разряды;
- сгенерировать исключение OverflowException.

По умолчанию при возникновении переполнения старшие разряды отбрасываются.

Можно явно задать режим контроля переполнения с помощью ключевых слов checked и unchecked. Ключевое слово checked задает режим контроля переполнения с генерацией исключения. Ключевое слово unchecked задает игнорирование возникновения переполнения.

Пример: Оператор ++ выполняется для переменной b с начальным значением 255:

- в блоке checked – возникает исключение, которое перехватывается в блоке catch;
- в блоке unchecked – исключение не возникает, в переменную b записывается значение 0.

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

```
{  byte b = 255;
   try
   {  checked
      { b++; //генерация OverflowException thrown }
        Console.WriteLine(b.ToString()); }
   catch (OverflowException e)
   { Console.WriteLine(e.Message);}
   try
   { unchecked
      { b++; //переполнение игнорируется }
      Console.WriteLine(b.ToString()); //0
   }
   catch (OverflowException e)
   { Console.WriteLine(e.Message);}
}
  }
}
```

Результат выполнения программы изображен на рисунке 2.13.

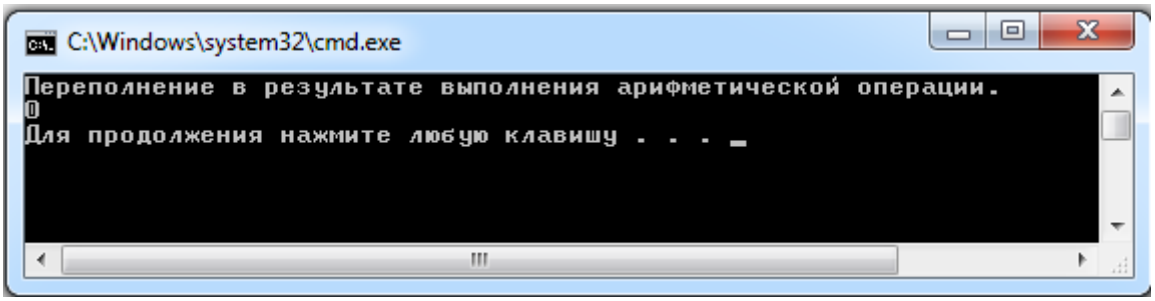

Рис. 2.13 Результат выполнения программы.

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?
42. Что понимается под термином «деструктор»?
43. В чем состоит назначение деструктора?
44. Приведите синтаксис деструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
45. Что такое рекурсия?
46. Что такое статистический метод?
47. Что такое перегрузка операторов?
48. Какие операторы можно перегружать?
49. Что такое индексаторы?
50. Для чего нужны одномерные и многомерные индексаторы?
51. Для чего нужны вложенные типы?
52. Для чего нужна обработка исключений?

53. Какие виды исключений бывают? Приведите примеры использования исключений в программах.
54. Для чего используются конструкции checked и unchecked для обработки исключений?

4 Задание

1. Написать программу в соответствии с вариантом задания. Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы; свойства, индексаторы; перегруженные операции. Функциональные элементы класса должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения. Описание каждого класса должны находиться в отдельном модуле. В проекте должна присутствовать диаграмма классов.
2. Отладить и протестировать программу.
3. Защитить лабораторную работу.
4. Базовый уровень, для студентов, претендующих на оценку 3. Повышенный для студентов, претендующих на оценку 4 или 5.

5 Варианты заданий

5.1 Базовый уровень

Варианты заданий определяются согласно списка студентов в группе.

Вариант 1.

Описать класс «Работник», содержащий поля:

– фамилия и инициалы работника;
– название занимаемой должности;
– зарплату;
– год поступления на работу.

Описать класс «Организация», содержащий поля:

– название организации;
– список работников (список объектов класса «Работник»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Работник»;
– вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры, если таких работников нет, вывести соответствующее сообщение;
– сортировку списка работников по фамилии и вывод отсортированного списка (для сравнения объектов в классе «Работник» перегрузить операции сравнения).

Вариант 2.

Описать класс с именем «Поезд», содержащий поля:

– название пункта назначения;
– номер поезда;
– время отправления.

Описать класс с именем «Станция», содержащий поля:

– название станции;
– список поездов, проходящих через станцию (список объектов класса «Поезд»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Поезд»;

– вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени, если таких поездов нет, вывести соответствующее сообщение;

– сортировку списка поездов по пункту назначения и вывод отсортированного списка (для сравнения объектов в классе «Поезд» перегрузить операции сравнения).

Вариант 3.

Описать класс с именем «Маршрут», содержащий поля:

– название начального пункта маршрута;

– название конечного пункта маршрута;

– номер маршрута.

Описать класс с именем «База отдыха», содержащий поля:

– название базы отдыха;

– список маршрутов (список объектов класса «Маршрут»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Маршрут»;

– вывод на экран информации о маршруте, номер которого введен с клавиатуры, если таких маршрутов нет, вывести соответствующее сообщение;

– сортировку списка маршрутов по конечному пункту и вывод отсортированного списка (для сравнения объектов в классе «Маршрут» перегрузить операции сравнения).

Вариант 4.

Описать класс с именем «Запись», содержащий поля:

– фамилия и имя;

– номер телефона;

– дата рождения (массив из трех чисел).

Описать класс с именем «Записная книжка», содержащий поля:

– название раздела;

– список записей (список объектов класса «Запись»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Запись»;

– вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры, если таких людей нет, вывести соответствующее сообщение;

– сортировку списка записей по фамилии и вывод отсортированного списка (для сравнения объектов в классе «Запись» перегрузить операции сравнения).

Вариант 5.

Описать класс с именем «Операция», содержащий поля:

– расчетный счет плательщика;

– расчетный счет получателя;

– перечисляемая сумма в руб.

Описать класс с именем «Журнал операций», содержащий поля:

– название банка;

– список операций (список объектов класса «Операция»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Операция»;

– вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры, если таких счетов нет, вывести соответствующее сообщение;
– сортировку списка операций по счету получателя и вывод отсортированного списка (для сравнения объектов в классе «Операция» перегрузить операции сравнения).

Вариант 6.
Описать класс с именем «Студент», содержащий поля:
– фамилия и инициалы;
– место рождения;
– успеваемость (массив из пяти элементов).
Описать класс с именем «Студенческая группа», содержащий поля:
– название группы;
– список студентов группы (список объектов класса «Студент»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Студент»;
– вывод на дисплей фамилий и номеров групп для всех студентов, если средний балл студента больше 4.0, если таких студентов нет, вывести соответствующие сообщение;
– сортировку списка студентов группы по месту рождения и вывод отсортированного списка (для сравнения объектов в классе «Студент» перегрузить операции сравнения).

Вариант 7.
Описать класс с именем «Рейс», содержащий поля:
– название пункта назначения рейса;
– номер рейса;
– тип самолета.
Описать класс с именем «Аэропорт», содержащий поля:
– название аэропорта;
– список рейсов (список объектов класса «Рейс»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Рейс»;
– вывод на экран информации о рейсе, номер которого введен с клавиатуры, если таких рейсов нет, вывести соответствующее сообщение;
– сортировку списка рейсов по пункту назначения и вывод отсортированного списка (для сравнения объектов в классе «Рейс» перегрузить операции сравнения).

Вариант 8.
Описать класс с именем «Книга», содержащий поля:
– название книги;
– фамилия и инициалы автора;
– год издания;
– категория.
Описать класс с именем «Библиотека», содержащий поля:
– название библиотеки;
– список книг (список объектов класса «Книга»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Книга»;

– вывод на экран информации о книге по фамилии автора, фамилия которого введена с клавиатуры, если таких книг нет, вывести соответствующее сообщение;
– сортировку списка книг по категории и вывод отсортированного списка (для сравнения объектов в классе «Книга» перегрузить операции сравнения).

Вариант 9.
Описать класс «Работник», содержащий поля:
– фамилия и инициалы работника;
– название занимаемой должности;
– зарплату;
– год поступления на работу.
Описать класс «Организация», содержащий поля:
– название организации;
– список работников (список объектов класса «Работник»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Работник»;
– вывод на дисплей фамилий работников, чья зарплата выше средней по организации и стаж работы больше трех лет, если таких работников нет, вывести соответствующее сообщение;
– сортировку списка работников по занимаемой должности и вывод отсортированного списка (для сравнения объектов в классе «Работник» перегрузить операции сравнения).

Вариант 10.
Описать класс с именем «Поезд», содержащий поля:
– название пункта назначения;
– номер поезда;
– время отправления.
Описать класс с именем «Станция», содержащий поля:
– название станции;
– список поездов, проходящих через станцию (список объектов класса «Поезд»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Поезд»;
– вывод на экран информации о пункте назначения, в который отправляется поезд, номер которого введен с клавиатуры, если таких поездов нет, вывести соответствующее сообщение.
– сортировку списка поездов по времени отправления и вывод отсортированного списка (для сравнения объектов в классе «Поезд» перегрузить операции сравнения).

Вариант 11.
Описать класс с именем «Маршрут», содержащий поля:
– название начального пункта маршрута;
– название конечного пункта маршрута;
– номер маршрута.
Описать класс с именем «База отдыха», содержащий поля:
– название базы отдыха;
– список маршрутов (список объектов класса «Маршрут»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Маршрут»;

– вывод на экран информации о маршруте, название конечного пункта которого введено с клавиатуры, если таких маршрутов нет, вывести соответствующее сообщение;
– сортировку списка маршрутов по начальному пункту и вывод отсортированного списка (для сравнения объектов в классе «Маршрут» перегрузить операции сравнения).

Вариант 12.
Описать класс с именем «Запись», содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).
Описать класс с именем «Записная книжка», содержащий поля:
– название раздела;
– список записей (список объектов класса «Запись»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Запись»;
– вывод на экран информации о людях, чьи дни рождения приходятся на год, значение которого введено с клавиатуры, если таких людей нет, вывести соответствующее сообщение;
– сортировку списка записей по дате рождения и вывод отсортированного списка (для сравнения объектов в классе «Запись» перегрузить операции сравнения).

Вариант 13.
Описать класс с именем «Студент», содержащий поля:
– фамилия и инициалы;
– место рождения;
– успеваемость (массив из пяти элементов).
Описать класс с именем «Студенческая группа», содержащий поля:
– название группы;
– список студентов группы (список объектов класса «Студент»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Студент»;
– вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2, если таких студентов нет, вывести соответствующие сообщение;
– сортировку списка студентов группы по фамилии и вывод отсортированного списка (для сравнения объектов в классе «Студент» перегрузить операции сравнения).

Вариант 14.
Описать класс с именем «Рейс», содержащий поля:
– название пункта назначения рейса;
– номер рейса;
– тип самолета.
Описать класс с именем «Аэропорт», содержащий поля:
– название аэропорта;
– список рейсов (список объектов класса «Рейс»).
Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа «Рейс»;
– вывод на экран информации о рейсе по типу самолета, название которого введено с клавиатуры, если таких рейсов нет, вывести соответствующее сообщение;

– сортировку списка рейсов по типу самолета и вывод отсортированного списка (для сравнения объектов в классе «Рейс» перегрузить операции сравнения).

Вариант 15.

Описать класс с именем «Книга», содержащий поля:

– название книги;
– фамилия и инициалы автора;
– год издания;
– категория.

Описать класс с именем «Библиотека», содержащий поля:

– название библиотеки;
– список книг (список объектов класса «Книга»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Книга»;
– вывод на экран информации о книге по категории, название которой введено с клавиатуры, если таких книг нет, вывести соответствующее сообщение;
– сортировку списка книг по году издания и вывод отсортированного списка (для сравнения объектов в классе «Книга» перегрузить операции сравнения).

Вариант 16.

Описать класс с именем «Карта», содержащий поля:

– масть;
– номер.

Описать класс с именем «Колода», содержащий поля:

– название библиотеки;
– список карт (список объектов класса «Карта»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Карта»;
– вывод на экран информации о карте по номеру, который был введен с клавиатуры, если таких номеров нет, вывести соответствующее сообщение;
– перемешивание колоды и выдача 6 карт случайным образом с сортировкой по масти и номеру и их вывод на экран (для сравнения объектов в классе «Карта» перегрузить операции сравнения).

Вариант 17.

Описать класс с именем «Поезд», содержащий поля:

– название пункта назначения;
– номер поезда;
– время отправления.

Описать класс с именем «Станция», содержащий поля:

– название станции;
– список поездов, проходящих через станцию (список объектов класса «Поезд»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Поезд»;
– вывод на экран информации о поездах, отправляющихся в пункт назначения введенного с клавиатуры, если таких поездов нет, вывести соответствующее сообщение.
– сортировку списка поездов по номеру поезда и вывод отсортированного списка (для сравнения объектов в классе «Поезд» перегрузить операции сравнения).

Вариант 18.

Описать класс с именем «Запись», содержащий поля:

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

Описать класс с именем «Записная книжка», содержащий поля:

– название раздела;
– список записей (список объектов класса «Запись»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Запись»;
– вывод на экран информации о людях, чьи дни рождения совпадают с введенными с клавиатуры, если таких людей нет, вывести соответствующее сообщение;
– сортировку списка записей по номеру телефона и вывод отсортированного списка (для сравнения объектов в классе «Запись» перегрузить операции сравнения).

Вариант 19.

Описать класс с именем «Студент», содержащий поля:

– фамилия и инициалы;
– место рождения;
– успеваемость (массив из пяти элементов).

Описать класс с именем «Студенческая группа», содержащий поля:

– название группы;
– список студентов группы (список объектов класса «Студент»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Студент»;
– вывод на дисплей фамилий и номеров групп для всех студентов, если они имеют оценки 4 и 5, если таких студентов нет, вывести соответствующие сообщение;
– сортировку списка студентов группы по среднему значению оценок и вывод отсортированного списка (для сравнения объектов в классе «Студент» перегрузить операции сравнения).

Вариант 20.

Описать класс с именем «Книга», содержащий поля:

– название книги;
– фамилия и инициалы автора;
– год издания;
– количество экземпляров.

Описать класс с именем «Библиотека», содержащий поля:

– название библиотеки;
– список книг (список объектов класса «Книга»).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных класса типа «Книга»;
– вывод на экран информации о книге количество экземпляров превышает значение введенное с клавиатуры, если таких книг нет, вывести соответствующее сообщение;
– сортировку списка книг по названию и вывод отсортированного списка (для сравнения объектов в классе «Книга» перегрузить операции сравнения).

5.2 Повышенный уровень

Варианты заданий определяет преподаватель.

Вариант 1.

Описать класс для работы с одномерным массивом целых чисел (вектором). Обеспечить следующие возможности:

- задание произвольных целых границ индексов при создании объекта;
- обращение к отдельному элементу массива с контролем выхода за пределы массива;
- выполнение операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов;
- выполнение операций умножения и деления всех элементов массива на скаляр;
- вывод на экран элемента массива по заданному индексу и всего массива.

Вариант 2.

Описать класс для работы с одномерным массивом строк фиксированной длины. Обеспечить следующие возможности:

- задание произвольных целых границ индексов при создании объекта;
- обращение к отдельной строке массива по индексу с контролем выхода за пределы массива;
- выполнение операций поэлементного сцепления двух массивов с образованием нового массива;
- выполнение операций слияния двух массивов с исключением повторяющихся элементов;
- вывод на экран элемента массива по заданному индексу и всего массива.

Вариант 3.

Описать класс многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Обеспечить следующие возможности:

- вычисление значения многочлена для заданного аргумента;
- операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена;
- получение коэффициента, заданного по индексу;
- вывод на экран описания многочлена.

Вариант 4.

Описать класс, обеспечивающий представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматрицы любого размера и всей матрицы, доступа по индексам к элементу матрицы, сложение и умножение элементов матрицы с заданным числом.

Вариант 5.

Описать класс для работы с восьмеричным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции:

- операции присваивания, реализующие значимую семантику;
- операции сравнения;
- преобразование в десятичное число;
- форматный вывод;
- доступ к заданной цифре числа по индексу.

Вариант 6.

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- сложение, вычитание (как с другой матрицей, так и с числом);
- комбинированные операции присваивания (+=, -=);
- операции сравнения на равенство/неравенство;
- операции вычисления обратной и транспонированной матрицы;
- доступ к элементу по индексам.

Вариант 7.

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- умножение, деление (как на другую матрицу, так и на число);
- комбинированные операции присваивания (*=, /=);
- операцию возведения в степень;
- методы вычисления определителя и нормы;
- доступ к элементу по индексам.

Вариант 8.

Описать класс для работы с двоичным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции:

- операции присваивания, реализующие значимую семантику;
- операции сравнения;
- преобразование в десятичное число;
- форматный вывод;
- доступ к заданной цифре числа по индексу.

Вариант 9.

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметричная, верхняя треугольная, нижняя треугольная);
- операции сравнения на равенство/неравенство;
- доступ к элементу по индексам.

Вариант 10.

Описать класс «предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых, это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя, сортировки по словам или по номеру страниц.

Вариант 11.

Описать класс «автостоянка» для хранения сведений об автомобилях. Для каждого автомобиля записываются госномер, цвет, фамилия владельца и признак присутствия на стоянке. Обеспечить возможность поиска автомобиля по разным критериям, вывода списка присутствующих и отсутствующих на стоянке автомобилей, доступа к имеющимся сведениям по номеру места, сортировки данных по различным полям. Класс автостоянка должен содержать список объектов класса «Автомобиль»

Вариант 12.

Описать класс для работы с четырехричным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции:

- операции присваивания, реализующие значимую семантику;
- операции сравнения;
- преобразование в десятичное число;
- форматный вывод;
- доступ к заданной цифре числа по индексу.

Вариант 13.

Описать класс «поезд», содержащий следующие закрытые поля:

- название пункта назначения;
- номер поезда (может содержать буквы и цифры);
- время отправления.

Предусмотреть свойства для получения состояния объекта.

Описать класс «вокзал», содержащий закрытый массив поездов. Для хранения списка вокзалов использовать стандартный список.

Обеспечить следующие возможности:

вывод информации о поезде по номеру с помощью индекса;

- вывод информации о поездах, отправляющихся после введенного с клавиатуры времени в определенный пункт назначения со всех вокзалов, информация должна быть отсортирована по времени отправления;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух поездов;
- вывод всех возможных маршрутов от одного вокзала до другого, как прямых, так и с пересадкой.

Вариант 14.

Описать класс «товар», содержащий следующие закрытые поля:

- название товара;
- название магазина, в котором продается товар;
- стоимость товара в рублях;
- количество товара.

Предусмотреть свойства для получения состояния объекта.

Описать класс «Справка», содержащий закрытый массив товаров. Обеспечить следующие возможности:

- • вывод информации о товаре по номеру с помощью индекса;
- • вывод на экран информации о товаре, название которого введено с клавиатуры, если таких товаров нет, выдать соответствующее сообщение;
- • сортировку товаров по названию магазина, по наименованию и по цене;
- • перегруженную операцию сложения товаров, выполняющую сложение их общей стоимости с группировкой по магазинам.

Вариант 15.

Описать класс «самолет», содержащий следующие закрытые поля:

- название пункта назначения;
- шестизначный номер рейса;
- время отправления.

Предусмотреть свойства для получения состояния объекта.

Описать класс «аэропорт», содержащий закрытый массив самолетов. Обеспечить следующие возможности:

- вывод информации о самолете по номеру рейса с помощью индекса;
- вывод информации о самолетах, отправляющихся в течение часа после введенного с клавиатуры времени;
- вывод информации о самолетах, отправляющихся в заданный пункт назначения, информация должна быть отсортирована по времени отправления;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух самолетов;
- вывод всех возможных маршрутов от одного аэропорта до другого, как прямых, так и с пересадкой.